



## **NetBurner's Runtime Libraries**

Revision 1.2  
July 11, 2007  
Released

## Table of Contents

|  |    |
|--|----|
| 1. Introduction.....                               | 10 |
| 2. NetBurner License Information .....             | 11 |
| 2.1. The NetBurner Tools Software License.....     | 11 |
| 2.2. The NetBurner Embedded Software License ..... | 11 |
| 2.3. Life Support Disclaimer .....                 | 12 |
| 2.4. Anti-Piracy Policy .....                      | 12 |
| 3. CAN Library.....                                | 13 |
| 3.1. Introduction.....                             | 13 |
| 3.2. CanRxMessage Class.....                       | 18 |
| 3.3. Constructors and Destructor.....              | 19 |
| 3.3.1. CanRxMessage - FIFO .....                   | 19 |
| 3.3.2. CanRxMessage - ID .....                     | 20 |
| 3.3.3. ~CanRxMessage .....                         | 21 |
| 3.4. Member Functions.....                         | 22 |
| 3.4.1. GetLength .....                             | 22 |
| 3.4.2. GetData .....                               | 23 |
| 3.4.3. GetId .....                                 | 24 |
| 3.4.4. GetTimeStamp .....                          | 25 |
| 3.4.5. IsValid.....                                | 26 |
| 3.5. Functions .....                               | 27 |
| 3.5.1. CanInit .....                               | 27 |
| 3.5.2. CanShutDown.....                            | 28 |
| 3.5.3. ChangeGlobalMask .....                      | 29 |
| 3.5.4. FreeCanChannels.....                        | 30 |
| 3.5.5. IsChannelFree.....                          | 31 |
| 3.5.6. RegisterCanRxFifo .....                     | 32 |
| 3.5.7. RegisterCanSpecialRxFifo.....               | 33 |
| 3.5.8. UnRegisterCanFifo.....                      | 34 |
| 3.5.9. SendMessage .....                           | 35 |
| 3.6. MACROS.....                                   | 36 |
| 3.6.1. CAN_EXTENDED_ID_BIT.....                    | 36 |
| 3.6.2. ExtToNbId .....                             | 37 |
| 3.6.3. NormToNbId.....                             | 38 |
| 3.6.4. IsNbIdExt .....                             | 39 |
| 3.6.5. NbToExtId .....                             | 40 |
| 3.6.6. NbToNormId.....                             | 41 |
| 4. Command Processor Library.....                  | 42 |
| 4.1. Introduction.....                             | 42 |
| 4.2. CmdStartCommandProcessor .....                | 43 |
| 4.3. CmdAddCommandFd.....                          | 44 |
| 4.4. CmdRemoveCommandFd.....                       | 45 |
| 4.5. CmdListenOnTcpPort.....                       | 46 |

|         |                                   |    |
|---------|-----------------------------------|----|
| 4.6.    | CmdStopListeningOnTcpPort .....   | 47 |
| 4.7.    | *CmdAuthenticateFunc .....        | 48 |
| 4.8.    | *CmdCmd_func.....                 | 49 |
| 4.9.    | *CmdConnect_func .....            | 50 |
| 4.10.   | *CmdPrompt_func.....              | 51 |
| 4.11.   | *CmdDisConnect_func .....         | 52 |
| 4.12.   | SendToAll .....                   | 53 |
| 4.13.   | Globals.....                      | 54 |
| 4.13.1. | CmdIdleTimeout.....               | 54 |
| 4.13.2. | *Cmdlogin_prompt .....            | 54 |
| 5.      | DHCP Library .....                | 55 |
| 5.1.    | Introduction.....                 | 55 |
| 5.2.    | Global Variables .....            | 56 |
| 5.3.    | StartDHCP .....                   | 57 |
| 5.4.    | StopDHCP .....                    | 58 |
| 5.5.    | RenewDHCP .....                   | 59 |
| 5.6.    | GetDHCPAddress .....              | 60 |
| 5.7.    | ValidDhcpLease.....               | 61 |
| 5.8.    | GetRemainingDhcpLeaseTime .....   | 62 |
| 5.9.    | PossiblyGetDHCPAddress .....      | 63 |
| 5.10.   | GetDHCPState .....                | 64 |
| 6.      | FTP Client Library .....          | 65 |
| 6.1.    | Introduction.....                 | 65 |
| 6.2.    | FTP_InitializeSession .....       | 69 |
| 6.3.    | FTP_CloseSession .....            | 70 |
| 6.4.    | FTPGetDir .....                   | 71 |
| 6.5.    | FTPSetDir .....                   | 72 |
| 6.6.    | FTPDeleteDir .....                | 73 |
| 6.7.    | FTPMakeDir .....                  | 74 |
| 6.8.    | FTPUpDir .....                    | 75 |
| 6.9.    | FTPDeleteFile .....               | 76 |
| 6.10.   | FTPRenameFile .....               | 77 |
| 6.11.   | FTPSendFile .....                 | 78 |
| 6.12.   | FTPGetFile .....                  | 80 |
| 6.13.   | FTPGetList.....                   | 82 |
| 6.14.   | FTPGetFileNames .....             | 84 |
| 6.15.   | FTPRawCommand .....               | 86 |
| 6.16.   | FTPGetCommandResult.....          | 87 |
| 6.17.   | FTPRawStreamCommand .....         | 88 |
| 7.      | FTP Server Library .....          | 89 |
| 7.1.    | Introduction.....                 | 89 |
| 7.2.    | FTPDStart .....                   | 91 |
| 7.3.    | FTPDStopReq .....                 | 92 |
| 7.4.    | ( FTPDCallBackReportFunct ) ..... | 93 |
| 7.5.    | FTPDSessionStart .....            | 94 |
| 7.6.    | FTPDSessionEnd.....               | 95 |

|       |  |     |
|-------|--|-----|
| 7.7.  | FTPD_DirectoryExists (User Defined).....     | 96  |
| 7.8.  | FTPD_CreateSubDirectory (User Defined) ..... | 97  |
| 7.9.  | FTPD_DeleteSubDirectory (User Defined).....  | 98  |
| 7.10. | FTPD_ListSubDirectories (User Defined) ..... | 99  |
| 7.11. | FTPD_FileExists (User Defined) .....         | 100 |
| 7.12. | FTPD_SendFileToClient (User Defined) .....   | 101 |
| 7.13. | FTPD_AbleToCreateFile (User Defined).....    | 102 |
| 7.14. | FTPD_GetFileFromClient (User Defined) .....  | 103 |
| 7.15. | FTPD_DeleteFile.....                         | 104 |
| 7.16. | FTPD_DeleteFile (User Defined) .....         | 105 |
| 7.17. | FTPD_ListFile .....                          | 106 |
| 7.18. | FTPD_ListFile (User Defined).....            | 107 |
| 7.19. | FTPD_Rename (User Defined) .....             | 109 |
| 8.    | HTTP and HTML Libraries .....                | 110 |
| 8.1.  | StartHTTP .....                              | 112 |
| 8.2.  | StopHTTP .....                               | 113 |
| 8.3.  | SetNewPostHandler.....                       | 114 |
| 8.4.  | SetNewGetHandler .....                       | 115 |
| 8.5.  | SetNewHeadHandler.....                       | 117 |
| 8.6.  | CheckAuthentication.....                     | 118 |
| 8.7.  | RequestAuthentication .....                  | 119 |
| 8.8.  | SendHTMLHeader .....                         | 120 |
| 8.9.  | SendHTMLHeaderWCookie .....                  | 121 |
| 8.10. | SendTextHeader .....                         | 122 |
| 8.11. | SendGifHeader .....                          | 123 |
| 8.12. | RedirectResponse.....                        | 124 |
| 8.13. | NotFoundResponse .....                       | 125 |
| 8.14. | ExtractPostData .....                        | 126 |
| 8.15. | ExtractPostFile .....                        | 127 |
| 8.16. | EnableMultiPartForms .....                   | 128 |
| 8.17. | DisableMultiPartForms .....                  | 129 |
| 8.18. | writesafestring .....                        | 130 |
| 8.19. | httpstricmp .....                            | 131 |
| 8.20. | SendFullResponse.....                        | 132 |
| 8.21. | SendFileFragment.....                        | 133 |
| 9.    | Interrupts.....                              | 134 |
| 9.1.  | INTERRUPT MACRO.....                         | 134 |
| 9.2.  | SetIntc (MCF5234 and MCF5282 Only).....      | 136 |
| 9.3.  | SetIntc (MCF5270 Only).....                  | 137 |
| 9.4.  | Example for MCF5234 and MCF5282 Only .....   | 138 |
| 10.   | I/O System Library .....                     | 139 |
| 10.1. | close.....                                   | 140 |
| 10.2. | read .....                                   | 141 |
| 10.3. | ReadWithTimeout.....                         | 142 |
| 10.4. | dataavail.....                               | 143 |
| 10.5. | charavail .....                              | 144 |

|         |   |     |
|---------|---|-----|
| 10.6.   | write.....                                | 145 |
| 10.7.   | Writestring .....                         | 146 |
| 10.8.   | writeall.....                             | 147 |
| 10.9.   | FD_ZERO .....                             | 148 |
| 10.10.  | FD_CLR.....                               | 149 |
| 10.11.  | FD_SET .....                              | 150 |
| 10.12.  | FD_ISSET .....                            | 151 |
| 10.13.  | select.....                               | 152 |
| 10.14.  | ZeroWaitSelect .....                      | 153 |
| 10.15.  | ioctl.....                                | 154 |
| 10.16.  | ReplaceStdio .....                        | 155 |
| 10.17.  | SysLog.....                               | 156 |
| 11.     | I <sup>2</sup> C Library.....             | 158 |
| 11.1.   | Introduction .....                        | 158 |
| 11.2.   | The NetBurner I <sup>2</sup> C API .....  | 160 |
| 11.3.   | Simple I <sup>2</sup> C Functions .....   | 162 |
| 11.3.1. | I2CInit .....                             | 162 |
| 11.3.2. | I2CSendBuf .....                          | 163 |
| 11.3.3. | I2CReadBuf .....                          | 164 |
| 11.3.4. | I2CRestart .....                          | 165 |
| 11.4.   | Slave I <sup>2</sup> C Functions.....     | 166 |
| 11.4.1. | I2CRXAvail .....                          | 166 |
| 11.4.2. | I2CTXAvail.....                           | 167 |
| 11.4.3. | I2CGetByte .....                          | 168 |
| 11.4.4. | I2CFillSlaveTXBuf.....                    | 169 |
| 11.5.   | Advanced I <sup>2</sup> C Functions ..... | 170 |
| 11.5.1. | I2CRead .....                             | 170 |
| 11.5.2. | I2CSend .....                             | 171 |
| 11.5.3. | I2CStart.....                             | 172 |
| 11.5.4. | I2CStop .....                             | 173 |
| 12.     | Multicast Library.....                    | 174 |
| 12.1.   | Introduction .....                        | 174 |
| 12.2.   | RegisterMulticastFifo.....                | 177 |
| 12.3.   | UnregisterMulticastFifo .....             | 178 |
| 13.     | NBTime Library.....                       | 179 |
| 13.1.   | Introduction .....                        | 179 |
| 13.2.   | time.....                                 | 180 |
| 13.3.   | set_time .....                            | 181 |
| 13.4.   | GetNTPTime .....                          | 182 |
| 13.5.   | SetNTPTime .....                          | 183 |
| 13.6.   | IOBoardRTCSetRTCfromSystemTime.....       | 184 |
| 13.7.   | IOBoardRTCSetSystemFromRTCTime.....       | 185 |
| 14.     | POP3 and E-Mail Libraries .....           | 186 |
| 14.1.   | Introduction .....                        | 186 |
| 14.2.   | POP3_InitializeSession.....               | 189 |
| 14.3.   | POP3_CloseSession .....                   | 190 |

|        |   |     |
|--------|---|-----|
| 14.4.  | POP3_StatCmd.....   | 191 |
| 14.5.  | POP3_ListCmd .....  | 192 |
| 14.6.  | POP3_DeleteCmd.....   | 193 |
| 14.7.  | POP3_RetrieveMessage .....  | 194 |
| 14.8.  | GetPOPErrorString .....   | 195 |
| 14.9.  | SendMail .....  | 196 |
| 14.10. | SendMailEx.....   | 197 |
| 14.11. | SendMailAuth .....  | 198 |
| 15.    | RTC Library .....   | 199 |
| 15.1.  | Introduction .....  | 199 |
| 15.2.  | RTCGetTime .....  | 200 |
| 15.3.  | RTCSetTime.....   | 201 |
| 15.4.  | RTCSetSystemFromRTCTime.....  | 202 |
| 15.5.  | RTCSetRTCfromSystemTime.....  | 203 |
| 15.6.  | SetNTPTime.....   | 204 |
| 16.    | Serial Library.....   | 205 |
| 16.1.  | Introduction .....  | 205 |
| 16.2.  | OpenSerial .....  | 206 |
| 16.3.  | SimpleOpenSerial .....  | 207 |
| 16.4.  | SerialClose .....   | 208 |
| 16.5.  | SerialEnableTxFlow .....  | 209 |
| 16.6.  | SerialEnableRxFlow .....  | 210 |
| 16.7.  | SerialEnableHwTxFlow.....   | 211 |
| 16.8.  | SerialEnableHwRxFlow .....  | 212 |
| 16.9.  | Serial485HalfDupMode .....  | 213 |
| 16.10. | SendBreak .....   | 214 |
| 16.11. | serwriteaddress .....   | 215 |
| 16.12. | GetUartErrorReg .....   | 216 |
| 16.13. | GetCD.....  | 217 |
| 16.14. | GetRI .....   | 218 |
| 16.15. | GetDSR .....  | 219 |
| 16.16. | SetDTR.....   | 220 |
| 17.    | SNMP Library .....  | 221 |
| 17.1.  | Introduction .....  | 221 |
| 17.2.  | SNMP Implementation Requirements .....  | 222 |
| 17.3.  | Level 0 --- Basic Instructions Using the SNMP Tools .....                       | 223 |
| 17.4.  | Level 1 --- Enable SNMP at the Absolute Minimum Level without Custom MIBs ..... | 224 |
| 17.5.  | Level 2 --- A Simple Custom MIB to Set/Clear Community Names<br>226             |     |
| 17.6.  | Note 1 --- Custom Tables.....   | 228 |
| 17.7.  | Note 2 --- Custom Community Name Parsing and Protection...                      | 230 |
| 17.8.  | Note 3 --- Traps and Custom Traps .....   | 231 |
| 17.9.  | Snmpget.....  | 232 |
| 17.10. | Snmpgetnext .....   | 233 |
| 17.11. | Snmpset.....  | 234 |

|        |  |     |
|--------|--|-----|
| 17.12. | Snmpwalk .....   | 235 |
| 18.    | SSL Library .....  | 236 |
| 18.1.  | Introduction .....   | 236 |
| 18.2.  | SSL Overview .....   | 238 |
| 18.3.  | Creating a Code Module for SSL Server Certificates .....               | 242 |
| 18.4.  | Creating a Code Module - SSL Server Key & Certificate - Diagram<br>246 |     |
| 18.5.  | Creating a Code Module for SSL Client Certificates .....               | 247 |
| 18.6.  | StartHTTPs .....   | 249 |
| 18.7.  | SSL_accept .....   | 250 |
| 18.8.  | IsSSLfd .....  | 251 |
| 18.9.  | SSL_GetSocketRemoteAddr .....  | 252 |
| 18.10. | SSL_GetSocketRemotePort .....  | 253 |
| 18.11. | SSL_GetSocketLocalAddr .....   | 254 |
| 18.12. | SSL_GetSocketLocalPort .....   | 255 |
| 18.13. | SSL_setsockopt .....   | 256 |
| 18.14. | SSL_clrsockopt .....   | 257 |
| 18.15. | SSL_getsockopt .....   | 258 |
| 18.16. | SSL_connect .....  | 259 |
| 19.    | Stream Update Library .....  | 260 |
| 19.1.  | Introduction .....   | 260 |
| 19.2.  | SendUserFlashToStreamAsBinary .....                                    | 261 |
| 19.3.  | SendUserFlashToStreamAsS19 .....                                       | 262 |
| 19.4.  | ReadS19UserFlashFromStream .....                                       | 263 |
| 19.5.  | ReadBinaryUserFlashFromStream .....                                    | 264 |
| 19.6.  | ReadS19ApplicationCodeFromStream .....                                 | 265 |
| 20.    | System Library .....   | 266 |
| 20.1.  | Constants .....  | 268 |
| 20.2.  | ConfigRecord .....   | 270 |
| 20.3.  | gConfigRec .....   | 271 |
| 20.4.  | Secs .....   | 272 |
| 20.5.  | TimeTick .....   | 273 |
| 20.6.  | Code Update Overview .....   | 274 |
| 20.7.  | Enable AutoUpdate .....  | 275 |
| 20.8.  | Update Shutdown Hook .....   | 276 |
| 20.9.  | Update Password Hook .....   | 277 |
| 20.10. | UpdateConfigRecord .....   | 278 |
| 20.11. | SetupDialog .....  | 279 |
| 20.12. | SaveUserParameters .....   | 280 |
| 20.13. | GetUserParameters .....  | 281 |
| 20.14. | putleds .....  | 282 |
| 20.15. | getdipsw .....   | 283 |
| 20.16. | ShowData .....   | 284 |
| 20.17. | ShowMac .....  | 285 |
| 20.18. | outbyte .....  | 286 |
| 20.19. | print .....  | 287 |

|          |                               |     |
|----------|-------------------------------|-----|
| 20.20.   | putnum.....                   | 288 |
| 20.21.   | AsciiTolp.....                | 289 |
| 20.22.   | ShowIP .....                  | 290 |
| 20.23.   | itoa.....                     | 291 |
| 20.24.   | ShowCounters.....             | 292 |
| 20.25.   | ForceReboot.....              | 293 |
| 20.25.1. | Example.....                  | 294 |
| 20.26.   | EtherLink .....               | 295 |
| 20.27.   | EtherSpeed100 .....           | 296 |
| 20.28.   | EtherDuplex.....              | 297 |
| 20.29.   | ManualEthernetConfig.....     | 298 |
| 21.      | TCP/IP Library .....          | 299 |
| 21.1.    | Introduction .....            | 299 |
| 21.2.    | InitializeStack .....         | 301 |
| 21.3.    | KillStack.....                | 302 |
| 21.4.    | AddInterface (Multihome)..... | 303 |
| 21.5.    | listen.....                   | 304 |
| 21.6.    | accept.....                   | 305 |
| 21.7.    | connect .....                 | 306 |
| 21.8.    | connectvia .....              | 307 |
| 21.9.    | setsockopt.....               | 308 |
| 21.10.   | clrsockopt .....              | 309 |
| 21.11.   | getsockopt .....              | 310 |
| 21.12.   | GetSocketRemoteAddr.....      | 311 |
| 21.13.   | GetSocketLocalAddr.....       | 312 |
| 21.14.   | GetSocketRemotePort .....     | 313 |
| 21.15.   | GetSocketLocalPort.....       | 314 |
| 21.16.   | GetHostByName.....            | 315 |
| 21.17.   | Ping .....                    | 316 |
| 21.18.   | PingViaInterface .....        | 317 |
| 21.19.   | SendPing .....                | 318 |
| 21.20.   | GetTFTP.....                  | 319 |
| 21.21.   | SendTFTP .....                | 320 |
| 21.22.   | ShowArp .....                 | 321 |
| 21.23.   | DumpTcpDebug .....            | 322 |
| 21.24.   | EnableTcpDebug.....           | 323 |
| 21.25.   | ShowIPBuffer .....            | 324 |
| 21.26.   | GetFreeCount.....             | 325 |
| 21.27.   | ShowBuffer .....              | 326 |
| 21.28.   | HTONS .....                   | 327 |
| 21.29.   | HTONL.....                    | 328 |
| 21.30.   | NTOHS .....                   | 329 |
| 21.31.   | NTOHL.....                    | 330 |
| 22.      | UDP Library.....              | 331 |
| 22.1.    | UDPPacket (FIFO) .....        | 333 |
| 22.2.    | UDPPacket (Pool Buffer) ..... | 334 |



|        |  |     |
|--------|--|-----|
| 22.3.  | UDPPacket .....                              | 335 |
| 22.4.  | ~UDPPacket .....                             | 336 |
| 22.5.  | Validate .....                               | 337 |
| 22.6.  | SetSourcePort .....                          | 338 |
| 22.7.  | GetSourcePort .....                          | 339 |
| 22.8.  | SetDestinationPort .....                     | 340 |
| 22.9.  | GetDestinationPort .....                     | 341 |
| 22.10. | GetDataBuffer .....                          | 342 |
| 22.11. | SetDataSize .....                            | 343 |
| 22.12. | GetDataSize .....                            | 344 |
| 22.13. | ResetData .....                              | 345 |
| 22.14. | AddData .....                                | 346 |
| 22.15. | AddData (Add a Zero Terminated String) ..... | 347 |
| 22.16. | AddDataWord .....                            | 348 |
| 22.17. | AddDataByte .....                            | 349 |
| 22.18. | ReleaseBuffer .....                          | 350 |
| 22.19. | GetPoolPtr .....                             | 351 |
| 22.20. | SendAndKeep .....                            | 352 |
| 22.21. | SendAndKeepVia .....                         | 353 |
| 22.22. | Send .....                                   | 354 |
| 22.23. | SendVia .....                                | 355 |
| 22.24. | RegisterUDPFifo .....                        | 356 |
| 22.25. | UnregisterUDPFifo .....                      | 357 |

# 1. Introduction

**Warning:** The Networking contents of this manual do not apply to any of NetBurner's Non-Network Development Kits (e.g. Mod5213). However, the CAN, I<sup>2</sup>C, and RTC Libraries apply to the MOD5213 Hardware Platform.

All NetBurner Non Network Development Kit User Manuals can be accessed directly from Windows: Start → Programs → NetBurner NNDK → Platform Hardware. They are located (by default) in your **C:\Nburn\docs\platform** directory.

**Warning:** The software included in your NNDK is licensed to run only on NetBurner provided hardware. If your application involves manufacturing your own hardware, please contact our [Sales](#) Department for details on a Royalty-Free Software License.

## Additional Documentation

- All NetBurner License Documentation is located by default in your C:\Nburn\docs directory.
- The NetBurner Documentation Overview is located by default in your C:\Nburn\docs directory.
- The NetBurner uCOS RTOS User's Manual is located by default in your C:\Nburn\docs directory.
- For specific NetBurner Hardware information, please refer to your Hardware User's Manual. From Windows: Start → Programs → NetBurner NNDK → Platform Hardware. By default, these manuals are located in your C:\Nburn\docs\Platform directory.
  - All NetBurner (public) Schematics are located by default in your C:\Nburn\docs\Platform\Schematics directory.
- All NetBurner PC Tools documentation is located by default in your C:\Nburn\docs directory.
- For additional Programming information, please refer to your NNDK Programmer's Guide. From Windows: Start → Programs → NetBurner NNDK → NNDK Programmers Guide. By default, this PDF is located in your C:\Nburn\docs directory.
- All Embedded Flash File System (EFFS) documentation (Mod5234, Mod5270, Mod5272, Mod5282, and PK70) is located by default in your C:\Nburn\docs directory.
- All Freescale documentation is located by default in your C:\Nburn\docs directory.
- All GNU documentation is located by default in your C:\Nburn\docs directory.
- For NBEclipse: Open up NBEclipse and from the Help pull down menu select Help Contents (Important: NBEclipse requires NNDK Version 2.0 or higher and Java Version 1.5 or higher on your host computer).

## 2. NetBurner License Information

The following three options require the purchase of a NetBurner License. If you have any questions, please contact our [Sales](#) Department.

1. NetBurner software running on your hardware
  2. NetBurner hardware reference design
  3. NetBurner hardware reference design and software
- All embedded software and source code provided in this Network Development Kit is subject to one of four possible licenses: the NetBurner Tools License (most restrictive), the NetBurner Embedded Software License, the GNU Public License and the Newlib License (least restrictive).
  - The GNU development executables provided in the C:\Nburn\GCC-M68k directory branch are subject to the GNU Public License (GPL).
  - The Runtime Libraries and include files provided in the C:\Nburn\GCC-M68k directory branch are subject to the Newlib License.
  - The Compcode application provided in the C:\Nburn\pctools\compcode directory is subject to the GNU public license (GPL).
  - All other programs are subject to the NetBurner Tools License provided below.
  - All other provided Source Code and Libraries are subject to the NetBurner Embedded Software License provided below.

### 2.1. The NetBurner Tools Software License

Copyright 1998 - 2007 NetBurner, Inc., All Rights Reserved.

1. Permission is hereby granted to purchasers of the NetBurner Network Development Kit to use these programs on one computer, and only to support the development of embedded applications that will run on NetBurner provided hardware.
2. No other rights to use this program or its derivatives, in part or in whole, are granted. It may be possible to license this or other NetBurner software for use on non NetBurner hardware.
3. NetBurner makes no representation or warranties with respect to the performance of this computer program, and specifically disclaims any responsibility for any damages, special or consequential, connected with the use of this program.

### 2.2. The NetBurner Embedded Software License

Copyright 1998 - 2007 NetBurner, Inc., All Rights Reserved.

1. Permission is hereby granted to purchasers of NetBurner hardware to use or modify this computer program for any use as long as the resultant program is only executed on NetBurner provided hardware.
2. No other rights to use this program or its derivatives, in part or in whole, are granted.
3. It may be possible to license this or other NetBurner software for use on non NetBurner hardware.

4. NetBurner makes no representation or warranties with respect to the performance of this computer program, and specifically disclaims any responsibility for any damages, special or consequential, connected with the use of this program.

### **2.3. Life Support Disclaimer**

NetBurner's products both hardware and software (including tools) are not authorized for use as critical components in life support devices or systems, without the express written approval of NetBurner, Inc. prior to use. As used herein:

1. Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. If you have any questions/concerns, please contact our Sales Department for more information.

### **2.4. Anti-Piracy Policy**

NetBurner, Inc. vigorously protects its copyrights, trademarks, patents and other intellectual property rights.

In the United States and many other countries, copyright law provides for severe civil and criminal penalties for the unauthorized reproduction or distribution of copyrighted material. Copyrighted material includes, but is not limited to computer programs and accompanying sounds, images and text.

Under U.S. law, infringement may result in civil damages of up to \$150,000, and/or criminal penalties of up to five years imprisonment, and/or a \$250,000 fine. In addition, NetBurner, Inc. may seek to recover its attorneys' fees.

## 3. CAN Library

### 3.1. Introduction

CAN or Controller Area Network is an advanced serial bus system that efficiently supports distributed control systems. CAN operates at data rates of up to 1 Megabit per second. CAN also has excellent error detection and confinement capabilities. CAN was initially developed in 1986 for the use in motor vehicles by Robert Bosch GmbH, in Germany, also holding the CAN license. For additional information, see the [CAN Homepage](#) of Robert Bosch GmbH. The first CAN silicon was fabricated in 1987 by Intel.

The CAN protocol is an international standard defined in ISO 11898 (for applications up to 1 Megabit per second) and ISO 11519 (for applications up to 125 Kilobits per second). The conformance test for the CAN protocol is defined in ISO 16845. ISO 16845 also guarantees the interchangeability of the CAN chips. See the [ISO](#) web site for additional information. CAN is also internationally standardized by the Society of Automotive Engineers ([SAE](#)).

The CAN communications protocol describes how information is passed between devices. CAN conforms to the Open Systems Interconnection (OSI) model (developed by the ISO - ISO 7498) which is defined in terms of layers. Each layer in a device "communicates" with the same layer in another device. Actual communication occurs between adjacent layers in each device. The devices are only connected by the physical medium via the physical layer of the OSI model.

The CAN architecture defines the lowest two layers of the OSI model - the Data Link Layer and the Physical Layer (lowest layer).

- The **Data Link Layer** is the **only** layer that recognizes and understands the format of messages. This layer constructs the messages to be sent to the Physical Layer, and decodes the messages received from the Physical Layer. In CAN controllers, the Data Link Layer is usually implemented in hardware.
- The **Physical Layer** specifies the physical and electrical characteristics of the CAN Bus, as well as the hardware that converts the characters of a message into electrical signals for transmitted messages, and electrical signals into characters for received messages. Although the other OSI layers may be implemented in either hardware (as chip level functions) or software, the Physical Layer is **always** "real" hardware.

The most common physical medium consists of a twisted wire pair (shielded or unshielded) with appropriate termination (i.e. 120 ohm resistance) at each end. ISO 11898 states that the impedance of the cable should be 120 +- 12 ohms. The basic CAN design specification called for a high bit rate, a high immunity to electrical interference, and an ability to detect any errors produced.

There are **no** standards for how CAN controllers are implemented, or how CAN controllers communicate with their host microcontroller.

The maximum bus length for a CAN network depends on the bit rate used. It is a requirement that the wave front of the bit signal have time to travel to the most remote node and back again (before the bit is sampled). So, if the bus length is near the maximum for the bit rate used (see the table on the next page), then the sampling point should be chosen very carefully.

| Bus Length (in meters) | Maximum Bit Rate (bits/second) |
|------------------------|--------------------------------|
| 40                     | 1 Mbit/s                       |
| 100                    | 500 Kbit/s                     |
| 200                    | 250 Kbit/s                     |
| 500                    | 125 Kbit/s                     |
| 6 km                   | 10 Kbit/s                      |

There are two principal CAN hardware implementations. Suitably configured, each implementation (Basic CAN and Full CAN) can handle both Normal and Extended CAN data formats. **Note:** Communication is identical for **all** implementations of CAN.

- **Basic CAN** is used in less expensive standalone CAN controllers or in smaller microcontrollers with an integrated CAN controller. In the Basic CAN configuration, there is a strong bond between the CAN controller and the associated microcontroller. Therefore, the microcontroller, which will have other system related functions to administer, will be interrupted to deal with every (received and transmitted) CAN message.
- **Full CAN** is used in more expensive, high performance CAN controllers and microcontrollers. Full CAN devices contain additional hardware to provide a separate "server" that will automatically receive and transmit CAN messages, without interrupting the associated microcontroller. Full CAN devices also service simultaneous requests, carry out extensive acceptance filtering on incoming messages, and greatly reduce the load on the microcontroller.

CAN identifiers come in two "flavors". These two "flavors" (i.e. protocol versions) define different formats of the message frame (with the main difference being the identifier length). Your Mod5234 supports both CAN versions. The two CAN protocol versions are:

- **Version 2.0A** - Normal or Standard CAN - supports messages with 11 bit identifiers.
- **Version 2.0B** - Extended CAN - supports messages with 29 bit identifiers (an 11 bit base identifier for compatibility with Version 2.0A and an 18 bit extension identifier).

In the NetBurner Software API, we **always** refer to **CAN identifiers** as **32 bit DWORDS**. A 32 bit DWORD is **bigger** than either (Normal or Extended) Identifier. A Normal identifier will **always** have bits 0 to 17 as zero. An Extended identifier **can have** bits 0 to 17 low. Extended identifiers that are received **will** have bit 29 set to 1. **Note:** Any ID input into the system will be treated as Extended **if** bit 29 is set, **or** if bits 0 to 17 are **not** zero.

There are **three** types of CAN controllers:

- **Part A** (Version 2.0A)
- **Part B Passive** (Version 2.0A)
- **Part B** (Version 2.0B)

Version 2.0B (i.e. Part B) controllers are **completely** backward compatible with Version 2.0A (both Part A and Part B Passive) controllers. Each CAN controller is able to handle the different parts of the CAN standard as shown on the next page.

| Message Format\CAN Chip Type             | Part A | Part B Passive         | Part B |
|--|--------|------------------------|--------|
| 11 bit ID (Normal or Standard) Ver. 2.0A | OK     | OK                     | OK     |
| 29 bit ID (Extended) Ver. 2.0B           | ERROR  | Tolerated– but ignored | OK     |

**Note:** If 29 bit identifiers are used on a CAN bus which contains Part A (Version 2.0A) controllers the bus will **not** work (as shown above). However, it is **possible** to use both Part B Passive (Version 2.0A) and Part B (Version 2.0B) controllers on a single network (also as shown above).

CAN is a multi-master Bus with an open linear structure, consisting of one logic bus line and equal nodes. The number of nodes is **not** limited by the protocol. In either of the two CAN protocols (i.e. Version 2.0A and Version 2.0B), the bus nodes do **not** have a specific address. Instead, the address information is contained in the identifiers of the transmitted messages - indicating both the message content and the priority of the message. Therefore, the number of nodes on a network may be changed dynamically **without** disturbing the communication of the other nodes.

A high degree of system and configuration flexibility is achieved because of CAN's content-oriented addressing scheme. Therefore, it is very easy to add additional stations (i.e. receivers) to an existing CAN network - without making any hardware or software modifications to the existing stations. This feature follows the concept of modular electronics, and permits multiple reception and synchronization of distributed processes. Data (needed as information by one or more stations) can be transmitted via the network in such a way that it is unnecessary for each station to know who produced it. This feature allows for the easy servicing and upgrading of networks (data transmission is not based on the availability of specific types of stations). Multicasting and Broadcasting are also supported by CAN.

There are two Bus states: Dominant and Recessive. The CAN Bus logic uses a "Wired-AND" mechanism. Dominant bits (logic 0) **overwrite** the recessive bits (logic 1). If **all nodes** on the network **transmit recessive bits** (Ones), the Bus is in the **recessive state**. However, as soon as **one** node **transmits** a **dominant bit** (Zero), the **Bus** state **changes** to **dominant**. A dominant state **will** always have precedence over a recessive state.

The CAN protocol handles Bus accesses according to an arbitration process known as Carrier Sense Multiple Access/Collision Detection (CSMA/CD). By using this non-destructive bitwise arbitration process the CAN Bus:

- Avoids collisions of messages whose transmission was started by more than one node simultaneously.
- Sends the most important message out first without time loss.

A message in the CAN Standard/Normal or Extended Frame format begins with a start bit called the Start Of Frame (SOF). This is a dominant bit for hard synchronization of all nodes on the network.

The Arbitration Field (12 bits) consists of the Message Identifier (11 bits) and the Remote Transmission Request (RTR) bit. The RTR bit is used to distinguish between a Data Frame (RTR bit is dominant) and a Remote Frame (RTR bit is recessive).

The Control Field (6 bits) contains the IDentifier Extension (IDE) bit (dominant to specify that the frame is a Standard Frame), a reserved dominant bit, and the Data Length Code (DLC) (4 bits). The DLC is used to indicate the number of data bytes in the Data Field. If the message is used as a Remote Frame, the DLC contains the number of requested data bytes. The Data Field that follows can hold up to 8 data bytes.

The Cyclic Redundancy Field is used to detect possible transmission errors. The integrity of the frame (Remote or Data) is guaranteed by the following Cyclic Redundant Check (CRC) sum. The CRC sum contains a 15 bit cyclic redundancy check code and a recessive delimiter bit.

The ACKnowledge (ACK) Field consists of two parts: the ACK Slot and the ACK Delimiter. The bit in the ACK Slot is initially sent as a recessive bit. This recessive bit is converted to a dominant bit by those receivers on the network that have received the data correctly. Correct messages are acknowledged by the receivers regardless of the result of the acceptance test (i.e. positive acknowledgement). The ACK Delimiter is also a recessive bit.

The end of the message is indicated by the End Of Frame (EOF) Field. This field contains seven recessive bits.

The Intermission Frame Space (IFS) Field follows the EOF. This is the minimum number of bits separating consecutive messages. After a three bit Intermission period, the Bus is recognized to be free. Note: The Bus Idle time may be any arbitrary length including zero.

A message in the CAN Extended Frame format (i.e. Version 2.0B) is almost the same as a message in the CAN Standard or Normal Frame format (i.e. Version 2.0A). One obvious difference is the length of the identifier used. The Extended Frame format identifier is made up of the existing 11 bit identifier (base identifier) and an 18 bit extension (identifier extension), for a total length of 29 bits. The distinction between the CAN Standard Frame format and the CAN Extended Frame format is made by using the IDE bit. The IDE bit is recessive to specify that the frame is an Extended Frame.

A Substitute Remote Request (SRR) bit is also included in the Arbitration Field (in Version 2.0B). The SRR bit is always transmitted as a recessive bit to ensure that, in the case of arbitration between a Standard Data Frame and an Extended Data Frame, the Standard Data Frame will always have priority if both messages have the same base (11 bit) identifier.

CAN provides superior error-detection and error handling mechanisms (e.g. a CRC check and a high immunity against electromagnetic interference). Erroneous messages are automatically retransmitted. Temporary errors are recovered. Permanent errors are followed by an automatic switch-off of defective nodes (stations). There is guaranteed system-wide data consistency.

CAN implements **five** different types of error detection - three at the message level and two at the bit level.

- At the **message level** the **three** types of error detection are:
  1. **Cyclic Redundancy Check (CRC)** - As already mentioned, the CRC safeguards the information in a frame by adding redundant check bits at the transmission end. At the receiving end, these bits are re-computed and tested against the received bits. If they do not match - a CRC error has occurred.
  2. **Frame Check** - This mechanism verifies the structure of the transmitted frame by checking the bit fields against the fixed format and the frame size. If they do not match - a frame check error has occurred. Errors detected by frame checks are called Format errors.



3. **ACK Errors** - As already mentioned, frames received are acknowledged by all receivers through positive acknowledgement. If no acknowledgement is received by the transmitter of the message - an ACK error has occurred.
- At the **bit level** the **two** types of error detection are:
    1. **Bit Monitoring** - The ability of the transmitter to detect errors is based on monitoring the CAN bus signals. Each transmitting station also observes the bus level, detecting differences between the bit sent and the bit received. This permits the reliable detection of global errors, and the detection of errors that are local to the individual transmitter.
    2. **Bit Stuffing** - The coding of the individual bits is tested at bit level. The bit representation used by CAN is called "Non Return to Zero" (NRZ) coding, which guarantees maximum efficiency in bit coding. The synchronization edges are generated by means of bit stuffing. This means that after five consecutive equal bits, the transmitter inserts into the bit stream a "stuff bit" with a complementary value which is removed by the receivers.

If one or more errors are discovered (by at least one station), the current (message) transmission is aborted by sending an "error flag". This error flag prevents other stations (on the same network) from accepting the message, and ensures the consistency of data throughout the network. After the transmission of an erroneous message (that has been aborted), the sender automatically re-attempts transmission (i.e. automatic re-transmission) of the message. However, in the event of a defective station, all messages (including valid ones) could be aborted.

Therefore, the CAN protocol also provides a mechanism to distinguish between sporadic errors, permanent errors and local failures at the station. This is accomplished by the statistical assessment of station error situations. The aim is - recognizing a station's (own) defects. Then, that station could switch to another mode, so that the rest of the CAN network is not negatively affected. For example, the defective station could switch itself off to prevent valid messages from erroneously being recognized as invalid.

A CAN higher level protocol (also known as the Application Layer) is a protocol implemented "on top" of the two existing lower-level CAN layers (i.e. the Physical Layer and the Data Link Layer). The application levels are linked to the physical medium by the layers of various emerging protocols, dedicated to a particular industry plus any number of propriety schemes as defined by individual CAN users.

Many systems (e.g. the automotive industry) use a propriety Application Layer; but for many other industries, this approach is not cost-effective. Several organizations have developed standardized open Application Layers to ensure ease of system integration. See the CAN in Automation ([CiA](#)) web site for additional information.

### Suggested Reading:

- CAN System Engineering: From Theory to Practical Applications by Wolfhard Lawrenz (ISBN - 0387949399)
- Controller Area Network by Konrad Etschberger (ISBN - 3000073760)

## 3.2. CanRxMessage Class

### Required Header File:

```
#include <canif.h>          // Found in C:\Nburn\<HWPlatform>\include
```

### Synopsis:

```
Class CanRxMessage
```

### Description:

The CanRxMessage Class will hold incoming messages. **Before** using this class, the application must have called CanInit, and be configured to **receive** incoming data on one or more ID's. There are two constructors that can be used to instantiate the class: using a FIFO and using an ID.

### The NetBurner Hardware Platforms that support CAN are:

- **CB34EX** (canif.h is located in C:\Nburn\CB34EX\include)
- **MOD5213** (canif.h is located in C:\Nburn\MOD5213\include)
- **MOD5234** (canif.h is located in C:\Nburn\MOD5234\include)
- **MOD5282** (canif.h is located in C:\Nburn\MOD5282\system)

### 3.3. Constructors and Destructor

#### 3.3.1. CanRxMessage - FIFO

##### Synopsis:

```
CanRxMessage( OS_FIFO * pFifo, WORD timeout );
```

##### Description:

This constructor builds a CanRxMessage from a FIFO. The FIFO **must be** registered to listen for incoming messages. The FIFO is a function of the RTOS. To use the FIFO, the application must:

1. Declare an OS\_FIFO object
2. Initialize the FIFO with the OSFifoInit function
3. Register the FIFO to listen to a specific ID
4. Create an instance of a CanRxMessage with the FIFO constructor

##### Parameters:

| Type    | Name    | Description  |
|---------|---------|--|
| OS_FIFO | * pFifo | A pointer to the FIFO used to communicate between the CAN subsystem and the CanRxMessage Class. The FIFO <b>must</b> be initialized first. |
| WORD    | timeout | How long to wait for confirmation. 0 = wait forever and 0xFFFF = don't wait at all.  |

##### Returns:

If **no** messages are received in the timeout interval (i.e. time ticks), then the returned CanRxMessage will be marked as **invalid**

A Timeout value of **0** (zero) **will** wait forever

**Note:** The **default** value is **1/20th** of a second

##### Example:

```
OS_FIFO fifo;
OSFifoInit( &fifo );
int chan = RegisterCanRxFifo( 0x1234, &fifo );
if ( chan > 0 )
{
    CanRxMessage can_msg( &fifo, 30*TICKS_PER_SECOND );
}
```

### 3.3.2. CanRxMessage - ID

#### Synopsis:

```
CanRxMessage( DWORD id, WORD timeout );
```

#### Description:

This constructor sends a RTR (Remote Transmission Request) to the device at a **specified** ID and waits for a response.

#### Parameters:

| Type  | Name    | Description   |
|-------|---------|---|
| DWORD | id      | The identifier to match on received frames.   |
| WORD  | timeout | How long to wait for confirmation. 0 = wait forever and 0xFFFF = don't wait at all. |

#### Returns:

If **no** messages are received in the timeout interval (i.e. time ticks), then the returned CanRxMessage will be marked as **invalid**

A Timeout value of **0** (zero) **will** wait forever. **Note:** The **default** timeout value is **1/20th of a second**.

The CAN system uses **any** unused channel to send and receive the buffer. This constructor can return an **invalid** message for **two** reasons:

1. The timeout interval has **transpired**
2. There were **no** free channels available to send the request

### 3.3.3. ~CanRxMessage

#### Synopsis:

```
~CanRxMessage( );
```

#### Description:

The CanRxMessage destructor

#### Parameters:

None

## 3.4. Member Functions

### 3.4.1. GetLength

#### Synopsis:

```
BYTE GetLength( );
```

#### Description:

This member function gets the amount of data stored in the message.

#### Parameters:

None

#### Returns:

The number of bytes stored as an unsigned 8 bit value

### 3.4.2. GetData

#### Synopsis:

**BYTE** GetData( **BYTE** \* buffer, **BYTE** max\_len );

#### Description:

This member function copies the data in a message object to the location pointed to by buffer up to a maximum of max\_len in bytes.

#### Parameters:

| Type | Name     | Description                                 |
|------|----------|---|
| BYTE | * buffer | A pointer to the buffer to put the data in. |
| BYTE | max_len  | The maximum length to store (in bytes).     |

#### Returns:

The number of bytes stored

### 3.4.3. GetId

#### Synopsis:

```
DWORD GetId( );
```

#### Description:

This member function gets the ID of the message. In the NetBurner Software API, we **always** refer to **CAN identifiers** as **32 bit DWORDS**. A 32 bit DWORD is **bigger** than either (Normal or Extended) Identifier.

A Normal identifier will **always** have bits 0 to 17 as zero. An Extended identifier **can have** bits 0 to 17 low. Therefore, Extended identifiers that are received **will** have bit 29 set to 1. **Note:** Any ID input into the system will be treated as Extended **if** bit 29 is set, **or** if bits 0 to 17 are **not** zero.

#### Parameters:

None

#### Returns:

The ID of the message object



### 3.4.4. GetTimeStamp

**Synopsis:**

```
WORD GetTimeStamp( );
```

**Description:**

Each CAN message contains a time stamp indicating when it is sent. This member function gets the time stamp from where it was sent.

**Parameters:**

None

**Returns:**

The time stamp of the message object

### 3.4.5. IsValid

#### Synopsis:

```
BOOL IsValid( );
```

#### Description:

Each CanRxMessage object constructor has a timeout value. If a message object is created and a timeout occurs the message object contains no data, and is marked as "invalid". This member function answers the question: Is this CanRxMessage a valid message.

#### Parameters:

None

#### Returns:

TRUE --- If the CanRxMessage object contains a **valid** message  
FALSE --- If the message is **invalid**

## 3.5. Functions

### 3.5.1. CanInit

#### Synopsis:

```
int CanInit( DWORD bit_rate, DWORD Global_Mask, BYTE irq_level=4 );
```

#### Description:

This function **initializes** the CAN system. It **must** be called **before** any other CAN functions **or** the creation of CanRxMessage objects.

#### Parameters:

| Type  | Name        | Description  |
|-------|-------------|--|
| DWORD | bit_rate    | The bit rate to run the CAN system at.                     |
| DWORD | Global_Mask | The mask used to mask received IDs.                        |
| BYTE  | irq_level   | The interrupt level you want the CAN system to operate at. |

**Note:** The system will get as close as possible, but 1000000, 500000, 250000, and 125000 are the **only** values that are known to work.

| Value | Meaning    |
|-------|------------|
| 0     | Don't care |
| 1     | Care       |

#### Returns:

CAN\_OK --- On success

CAN\_RATE\_FAIL --- If the bit rate could not be set within 1.5%

CAN\_ALREADYOPEN --- If the CAN system is already running - you **must** call CanShutDown first

### 3.5.2. CanShutDown

#### Synopsis:

```
void CanShutDown( );
```

#### Description:

This function shuts down the CAN system.

#### Parameters:

None

#### Returns:

Nothing --- This is a void function

### 3.5.3. ChangeGlobalMask

#### Synopsis:

```
void ChangeGlobalMask( DWORD Global_Mask );
```

#### Description:

This function **changes** the global receive mask **after** the CAN system is started.

#### Parameter:

| Type  | Name        | Description                         |
|-------|-------------|-------------------------------------|
| DWORD | Global_Mask | The mask used to mask received IDs. |

#### Returns:

Nothing --- This is a void function

### 3.5.4. FreeCanChannels

**Synopsis:**

```
int FreeCanChannels( );
```

**Description:**

The CAN system has 16 available channels. Use this function return to determine which channels are currently free.

**Parameters:**

None

**Returns:**

The number of channels that are currently not in use

### 3.5.5. IsChannelFree

**Synopsis:**

```
BOOL IsChannelFree( int channel );
```

**Description:**

This function tells you if a specific channel is currently free.

**Parameter:**

| Type | Name    | Description            |
|------|---------|------------------------|
| int  | channel | The specified channel. |

**Returns:**

TRUE --- If the specified channel is currently free

### 3.5.6. RegisterCanRxFifo

#### Synopsis:

```
int RegisterCanRxFifo( DWORD id, OS_FIFO * pFifo, int channel=-1 );
```

#### Description:

This function tells the CAN system to start listening for a specific CAN ID. Any incoming CAN frames that match the ID as set by the appropriate mask will be placed into the FIFO. The FIFO is a function of the RTOS. To use the FIFO, the application **must**:

1. Declare an OS\_FIFO object
2. Initialize the FIFO with the OSFifoInit function.
3. Register the FIFO to listen to a specific ID.
4. Create an instance of a CanRxMessage with the FIFO constructor.

#### Parameters:

| Type    | Name    | Description   |
|---------|---------|---|
| DWORD   | id      | The identifier to match on received frames. The id is modified by the global mask.  |
| OS_FIFO | * pFifo | A pointer to the FIFO, used to communicate between the CAN subsystem and the CanRxMessage Class. The FIFO <b>must</b> be initialized first. The same FIFO can be passed to multiple receive registration functions. |
| int     | channel | There are a total of 16 CAN channels. You can either specify a channel to use for the receive request, or you can specify a value of -1, which allows the system to select an unused channel.                       |

#### Returns:

A value 0 to 15 --- The channel this request is assigned to.

**Note:** This value **must** be stored to later call **UnRegisterCanFifo**

CAN\_CHANNEL\_USED --- If the channel is used or there are **no** free channels

#### Example:

```
OS_FIFO fifo;
OSFifoInit( &fifo );
int chan = RegisterCanRxFifo( 0x1234, &fifo );
if (chan > 0)
{
    CanRxMessage can_msg( &fifo, 30*TICKS_PER_SECOND );
}
```



### 3.5.7. RegisterCanSpecialRxFifo

#### Synopsis:

```
int RegisterCanSpecialRxFifo( DWORD id, DWORD spl_mask, OS_FIFO *
pFifo, int channel=-1 );
```

#### Description:

This function instructs the CAN system to start listening for a **specific** CAN ID. Any incoming CAN frames that match the ID (as set by the appropriate mask) will be placed into the FIFO.

**Note:** Some applications may require **more** than one channel mask. The NetBurner CAN device can have up to 3 masks:

1. The global mask for channels 0 -13
2. A mask for channel 14
3. A mask for channel 15

**Note:** The masks for channels 14 and 15 are set using the spl\_mask parameter unique to this function.

#### Parameters:

| Type   | Name     | Description   |
|--------|----------|---|
| DWORD  | id       | The identifier to match on received frames. This is modified by the passed in mask.   |
| DWORD  | spl_mask | There are only <b>two</b> channels available for use with the special mask so use this call sparingly and only if really needed.  |
| OSFifo | * pFifo  | A pointer to the FIFO, used to communicate between the CAN subsystem and the CanRxMessage Class. The FIFO <b>must</b> be initialized first. The same FIFO can be passed to multiple receive registration functions. |
| int    | channel  | There are a total of 16 CAN channels. You can either specify a channel to use for the receive request, or you can specify a value of -1, which allows the system to select an unused channel.                       |

#### Returns:

A value 0 to 15 --- The channel this request is assigned to.

**Note:** This value **must** be **stored** to later call **UnRegisterCanFifo**

CAN\_CHANNEL\_USED --- If the channel is used or there are no free channels

### 3.5.8. UnRegisterCanFifo

#### Synopsis:

```
int UnRegisterCanFifo( int channel );
```

#### Description:

This function disconnects a receiver channel from a FIFO.

#### Parameter:

| Type | Name    | Description            |
|------|---------|------------------------|
| int  | channel | The channel to remove. |

#### Returns:

CAN\_OK --- If successful

CAN\_CHANNEL\_NOT\_USED --- If the channel was **not** currently in use

### 3.5.9. SendMessage

#### Synopsis:

```
int SendMessage( DWORD id, BYTE * data, BYTE len, WORD timeout, int  
channel = -1 );
```

#### Description:

This function sends a message to a device with the specified id. To send a message, one of the 16 channels **must** be available. **Note:** The channel will **automatically** be freed once the message has been sent.

#### Parameters:

| Type  | Name    | Description  |
|-------|---------|--|
| DWORD | id      | The identifier to send.  |
| BYTE  | * data  | A pointer to the data to send.   |
| BYTE  | len     | The length of the data. It <b>must</b> be less than or equal to 8 bytes.   |
| WORD  | timeout | How long to wait for confirmation it sent. 0 = wait forever.<br>0xFFFF = don't wait at all. Any other timeout value blocks until<br>it is actually sent. |
| int   | channel | The channel to use. A value of -1 will allow the system to select<br>an unused channel.  |

#### Returns:

CAN\_OK --- If the message was sent

CAN\_CHANNEL\_USED --- Can't send because the channel was already in use or no channels  
available

CAN\_TIMEOUT --- Did not send in the time allotted

## 3.6. MACROS

### 3.6.1. CAN\_EXTENDED\_ID\_BIT

#### Synopsis:

```
#define CAN_EXTENDED_ID_BIT( 0x20000000 )
```

#### Description:

This macro takes the single bit used by the API to indicate an extended ID.

### 3.6.2. ExtToNbId

#### Synopsis:

```
#define ExtToNbId( id ) ( id | CAN_EXTENDED_ID_BIT )
```

#### Description:

This macro will make a system recognized Extended ID from either an Extended (29 bit) CAN Identifier or from a Normal (11 bit) CAN Identifier.

### 3.6.3. NormToNbId

#### Synopsis:

```
#define NormToNbId( id ) ( ( id & 0x7ff )<<18 )
```

#### Description:

This macro creates a Normal ID, and also an ID set from a normal id in the range 0 to 2048.

### 3.6.4. IsNBIdExt

#### Synopsis:

```
#define IsNBIdExt( id ) ( ( id & ( CAN_EXTENDED_ID_BIT|0x3FFFF ) )!=0 )
```

#### Description:

This macro determines if the ID is extended.

### 3.6.5. NbToExtId

#### Synopsis:

```
#define NbToExtId( id ) ( id & 0x1FFFFFFF )
```

#### Description:

This macro **strips** the extra flag. This macro **removes** the API extended flag from the ID.



### 3.6.6. NbToNormId

#### Synopsis:

```
#define NbToNormId( id ) ( ( id >> 18 ) & 0x7FF )
```

#### Description:

This macro will **shift** a Normal ID so that it has a value 0 to 1023. Some CAN systems will treat normal ID's as an integer from 0 to 2048. Other CAN systems may treat normal IDs as 28 bit values where the bottom 17 bits are zero. This macro **will** convert our Normal ID format into the 0 to 2048 format.

## 4. Command Processor Library

### 4.1. Introduction

The Command Processor is a system program that accepts user commands and converts them into the machine commands required by the operating system. The Command Processor receives and executes operating system commands. After you enter a command, the command processor analyzes the syntax to make sure the command is valid, and then either executes the command or issues an error warning.

### Required Header File

```
#include <command.h>    // Found in C:\Nburn\include
```

### Functions

- CmdStartCommandProcessor --- Starts the command processor
- CmdAddCommandFd --- Adds an established fd connection to the list of fds managed by the command processor
- CmdRemoveCommandFd --- Removes an established fd
- CmdListenOnTcpPort --- Listens for a connection on a TCP port
- CmdStopListeningOnTcpPort --- Stops listening for connections on the specified port
- \*CmdAuthenticateFunc --- Authenticates the username and password
- \*CmdCmd\_func --- Processes a command
- \*CmdConnect\_func --- Called whenever a new connection is established
- \*CmdPrompt\_func --- Called to display a command prompt
- \*CmdDisconnect\_func --- Called whenever a command is disconnected
- SendToAll --- Send to all connected sockets, excluding "Listening" sockets

### Globals

- CmdIdleTimeout
- \*Cmdlogin\_prompt

## 4.2. CmdStartCommandProcessor

### Synopsis:

```
int CmdStartCommandProcessor( int priority );
```

### Description:

This function starts the Command Processor.

### Parameter:

| Type | Name     | Description                       |
|------|----------|-----------------------------------|
| int  | priority | The UCOS task priority to run at. |

### Returns:

CMD\_OK --- On Success

CMD\_FAIL --- On Failure

### Example:

The newdemo application --- Located by default in your C:\Nburn\examples directory.

**Warning:** This application will not run on a SB72

### 4.3. CmdAddCommandFd

#### Synopsis:

```
int CmdAddCommandFd( int fd, int require_auth, int time_out_conn, int  
local_echo=TRUE );
```

#### Description:

This function adds an established file descriptor (fd) connection to the list of file descriptors managed by the command processor.

#### Parameters:

| Type | Name            | Description   |
|------|-----------------|---|
| int  | fd              | The file descriptor                                       |
| int  | require_auth    | Do we authenticate the connection on this file descriptor |
| int  | time_out_conn   | Do we time out the connection on this file descriptor     |
| int  | local_echo=TRUE | The command processor will echo the file descriptor.      |

#### Returns:

CMD\_OK --- On Success

CMD\_FAIL --- On Failure

CMD\_TO\_MANY\_FDS --- If there are too many file descriptors

## 4.4. CmdRemoveCommandFd

### Synopsis:

```
int CmdRemoveCommandFd( int fd );
```

### Description:

This function removes an established file descriptor (either a TCP or a Serial connection).

### Parameter:

| Type | Name | Description         |
|------|------|---------------------|
| int  | fd   | The file descriptor |

### Returns:

CMD\_OK --- On Success

CMD\_FAIL --- On Failure

## 4.5. CmdListenOnTcpPort

### Synopsis:

```
int CmdListenOnTcpPort( WORD port, int do_telnet_processing, int  
max_connections );
```

### Description:

This function starts listening for a connection on a TCP port. Telnet is a user command and an underlying TCP/IP protocol for accessing remote computers.

### Parameters:

| Type | Name                 | Description  |
|------|----------------------|--|
| WORD | port                 | The port number  |
| int  | do_telnet_processing | Should we treat the port as Telnet and process a Telnet negotiation? |
| int  | max_connections      | What is the maximum number of connections allowed on this port?      |

### Returns:

CMD\_OK --- On Success  
CMD\_FAIL --- On Failure

## 4.6. CmdStopListeningOnTcpPort

### Synopsis:

```
int CmdStopListeningOnTcpPort( WORD port );
```

### Description:

This function stops listening for connections on the specified port. **Note:** This function also **closes** all **open** connections that were based on that **specified** port.

### Parameter:

| Type | Name | Description     |
|------|------|-----------------|
| WORD | port | The port number |

### Returns:

CMD\_OK --- On Success  
CMD\_FAIL --- On Failure

## 4.7. \*CmdAuthenticateFunc

### Synopsis:

```
extern int ( * CmdAuthenticateFunc )( const char * name, const char *  
passwd );
```

This function is of the form:

```
int AuthenticateCommand ( const char * name, const char * passwd );
```

### Description:

This external authentication function CALLBACK is used to verify the Username and Password.

**Note:** If this function pointer is **not NULL**, then each new Telnet session **will** be asked to **authenticate** the Username and Password.

### Parameters:

| Type       | Name    | Description                |
|------------|---------|----------------------------|
| const char | *name   | A pointer to the Username. |
| const char | *passwd | A pointer to the Password. |

### Returns:

CMD\_OK --- If the authentication was OK

CMD\_CLOSE --- If the authentication causes the session to terminate (i.e. close)



## 4.8. \*CmdCmd\_func

### Synopsis:

```
extern int ( * CmdCmd_func )( const char *command, FILE *fRespondto,  
void *pData );
```

This function is of the form:

```
int ProcessCommand( const char * command, int fd_respondeto );
```

### Description:

This is the command processing CALLBACK function.

### Parameters:

| Type       | Name        | Description  |
|------------|-------------|--|
| const char | *command    | A pointer to the NULL terminated ASCII text of the command.  |
| FILE       | *fRespondto | The file descriptor (fd) that <b>all</b> response should be sent to.<br><b>Note:</b> Use fprintf or fiprintf to <b>write</b> to the file descriptor. |
| void       | *pData      | The pointer to a data object.  |

### Returns:

CMD\_OK --- If the command was OK

CMD\_CLOSE --- If the command causes the session to terminate (i.e. close)

## 4.9. \*CmdConnect\_func

### Synopsis:

```
extern void* ( * CmdConnect_func )( FILE *fRespondto );
```

This function is of the form:

```
void ConnectCommand ( FILE * fRespondto );
```

### Description:

This is the connect CALLBACK function. **Note:** If this function pointer is **not NULL**, then the system **will** call this function **every** time a new session is **started**.

### Parameter:

| Type | Name        | Description   |
|------|-------------|---|
| FILE | *fRespondto | The file descriptor (fd) that <b>all</b> response should be sent to.<br><b>Note:</b> Use fprintf <b>or</b> fiprintf to <b>write</b> to the file descriptor. |

### Returns:

An arbitrary void \* data item to be associated with this session

## 4.10. \*CmdPrompt\_func

### Synopsis:

```
extern void ( * CmdPrompt_func )( FILE *fRespondto, void *pData );
```

This function is of the form:

```
void PromptCommand( int fd_respondeto )
```

### Description:

This is a prompt Callback function. **Note:** If this function pointer is **not** NULL, then the system **will** call this function **every** time a **new** prompt line **needs** to be displayed.

### Parameters:

| Type | Name        | Description  |
|------|-------------|--|
| FILE | *fRespondto | The file descriptor (fd) that <b>all</b> response should be sent to.<br><b>Note:</b> Use fprintf <b>or</b> fiprintf to <b>write</b> to the file descriptor.  |
| void | *pData      | The pointer to a data object that can be used in the prompt. The value of pData is the value returned by *CmdConnect_func. This means that you can change the value of pData for each connection, and make that value part of the command prompt (e.g. NB:1>, NB:2>, NB:3>, etc.). |

### Returns:

Nothing --- This is a void function

### Example:

```
#include <command.h>
void ProcessPrompt( FILE *fp, void *pData )
{
    // The following code will produce the prompt: "MyPrompt> "
    fiprintf( fp, "\nMyPrompt> " );
}
void UserMain( void *pd )
{
    ...
    CmdPrompt_func = ProcessPrompt;
    ...
}
```

## 4.11. \*CmdDisconnect\_func

### Synopsis:

```
extern void ( * CmdDisconnect_func )( FILE *fRespondto, int cause, void
*pData );
```

This function is of the form:

```
void DisconnectCommand( int fd_respondeto, int cause );
```

### Description:

This is a disconnect CALLBACK function. Note: If this function pointer is not NULL, then the system will call this function every time a session is terminated.

### Parameters:

| Type | Name        | Description   |
|------|-------------|---|
| FILE | *fRespondto | The file descriptor (fd) that all responses should be sent to. <b>Note:</b> Use fprintf <b>or</b> fiprintf to <b>write</b> to the file descriptor.  |
| int  | cause       | The reason why it is disconnected. The current values of cause are: <ul style="list-style-type: none"> <li>• #define CMD_DIS_CAUSE_TIMEOUT (1)</li> <li>• #define CMD_DIS_CAUSE_CLOSED (2)</li> <li>• #define CMD_DIS_SOCKET_CLOSED (3) --- Do not send a response for this case.</li> <li>• #define CMD_DIS_AUTH_FAILED (4) --- Do not send a response for this case.</li> </ul> |
| void | *pData      | The pointer to a data object.   |

### Returns:

Nothing --- This is a void function

## 4.12. SendToAll

### Synopsis:

```
void SendToAll ( char *buffer, int len, BOOL include_serial_ports );
```

### Description:

This function will send to all connected sockets, excluding "Listening sockets".

### Parameters:

| Type | Name                 | Description  |
|------|----------------------|--|
| char | *buffer              | Pointer to the buffer.                                 |
| int  | len                  | The length of what is in the buffer.                   |
| BOOL | include_serial_ports | Do you want to include Serial ports or just TCP ports? |

### Returns:

Nothing --- This is a void function

## 4.13. Globals

### 4.13.1. CmdIdleTimeout

#### Synopsis:

```
extern int CmdIdleTimeout;
```

#### Description:

The number of seconds a connection is idle before it is terminated due to inactivity.

### 4.13.2. \*Cmdlogin\_prompt

#### Synopsis:

```
extern const char *Cmdlogin_prompt;
```

#### Description:

If this is not NULL, then it will be sent to the socket on connection (before authentication is tried).

## 5. DHCP Library

### 5.1. Introduction

This Library provides a DHCP Client to dynamically get IP Addresses using the RFC1541 DHCP protocol. **Important:** To use this Library, you **must** call the StartDHCP function **after** the IP stack is initialized. **Note:** The TicTacToe example program (located by default in C:\Nburn\examples) uses this capability.

#### Required Header File

```
#include <dhcpcclient.h>    // Found in C:\Nburn\include
```

#### DHCP Client Variables

- Global Variables

#### DHCP Client Operational Functions

- StartDHCP --- Starts the DHCP client
- StopDHCP --- Stops the DHCP client, and release any DHCP leases we own
- RenewDHCP --- Forces a DHCP renew
- GetDHCPAddress --- The simplest way to use DHCP
- ValidDhcpLease --- Determines the validity of the DHCP lease
- GetRemainingDhcpLeaseTime --- Retrieves the number of seconds remaining on the current DHCP lease
- PossiblyGetDHCPAddress --- Checks the status of the interface IP and gets a DHCP address if needed

#### DHCP Status Reporting Function

- GetDHCPState --- Get the current DHCP state

## 5.2. Global Variables

| Type           | Name            | Description   |
|----------------|-----------------|---|
| IPADDR         | DhcpClientIP    | The allocated IP Address.   |
| IPADDR         | DhcpClientMask  | The allocated Subnet Mask.  |
| IPADDR         | DhcpServerIP    | The Server ID.  |
| IPADDR         | DhcpRelayIP     | The Relay Agent.  |
| IPADDR         | DhcpRouterIP    | The Gateway IP Address.   |
| IPADDR         | DhcpDNSIP       | The DNS IP Address.   |
| Volatile DWORD | DhcpLeaseTimer  | Tracks the current lease time.  |
| DWORD          | DhcpLeaseStart  | IP Address lease start time in seconds.   |
| DWORD          | DhcpLeaseTime   | IP Address lease time in seconds.   |
| DWORD          | DhcpRenewTime   | Time to renewing state in seconds.  |
| DWORD          | DhcpRebindTime  | Time to rebinding state in seconds.   |
| const char     | *pDHCPOfferName | The name to give this DHCP client.<br><b>Note:</b> This name <b>must</b> be set before calling StartDHCP. |



## 5.3. StartDHCP

### Synopsis:

```
void StartDHCP( OS_SEM * pDhcpSemaphore );
```

### Description:

This function starts the DHCP client. This function will start the DHCP process. If you point the DHCP global variable pDHCPOfferName to a name, the DHCP system will assign that name. It will return immediately, and you need to either watch the semaphore, or monitor the status of DHCP using the GetDHCPState function before you assume that the IP Address has been setup.

### Parameter:

| Type   | Name            | Description   |
|--------|-----------------|---|
| OS_SEM | *pDhcpSemaphore | A pointer to a Semaphore - to notify when DHCP is complete. <b>Note:</b> It may be passed in as <b>NULL</b> . |

### Returns:

Nothing --- This is a void function

### Example Code:

```
#include <dhcpclient.h>
*
*
*
{
OS_SEM DHCPSem;
OSSemInit(&DHCPSem,0); // Initialize the Semaphore
StartDHCP(&DHCPSem); // Start DHCP
if (OSSemPend(&dhcpSem,20*TICKS_PER_SECOND)==OS_TIMEOUT) // Wait 20 sec
    { // DHCP did not initialize, handle the error here
    }
}
```

## 5.4. StopDHCP

### Synopsis:

```
void StopDHCP( );
```

### Description:

This function stops the DHCP client **and** releases any DHCP leases we own.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 5.5. RenewDHCP

### Synopsis:

```
void RenewDHCP( );
```

### Description:

DHCP uses the concept of a "lease" or amount of time that a given IP Address will be valid for a computer. Therefore, calling this function **will** result in the "lease" being renewed (i.e. a **forced** renew of the lease).

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 5.6. GetDHCPAddress

### Synopsis:

```
int GetDHCPAddress( );
```

### Description:

This function encapsulates **all** of the code necessary to start the DHCP Client.

### Parameters:

None

### Return Values:

DHCP\_OK (0) --- The system found a DHCP address

DHCP\_FAILED (-1) --- The system failed to acquire a DHCP address

### Example:

```
// Beginning of an application using DHCP
void UserMain(void * pd)
{
    InitializeStack();
    if ((EthernetIP==0) && (GetDHCPAddress()!=DHCP_OK))
    {
        iprintf("Failed to get DHCP address\r\n");
        // Should do error handling here, what to do if no address is available
    }
    OSChangePrio(MAIN_PRIO);
    EnableAutoUpdate();
    *
    *
    UserMain continues...
```

## 5.7. ValidDhcpLease

### Synopsis:

```
BOOL ValidDhcpLease( );
```

### Description:

DHCP uses the concept of a "lease" or amount of time that a given IP Address will be valid for a computer. Therefore, call this function to determine the validity of the DHCP "lease".

### Parameters:

None

### Returns:

1 - TRUE --- If the "lease" is valid  
0 - FALSE --- If the "lease" is not valid

## 5.8. GetRemainingDhcpLeaseTime

### Synopsis:

```
DWORD GetRemainingDhcpLeaseTime( );
```

### Description:

DHCP uses the concept of a "lease" or amount of time that a given IP address will be valid for a computer. Therefore, call this function to retrieve the number of seconds remaining for the current valid "lease".

### Parameters:

None

### Returns:

The amount of times left (in seconds) on the current DHCP "lease"

## 5.9. PossiblyGetDHCPAddress

### Synopsis:

```
int PossiblyGetDHCPAddress( int interface = 0 );
```

### Description:

This function checks the status of the interface IP. Note: If none is configured, it starts DHCP.

### Parameter:

| Type | Name          | Description                           |
|------|---------------|---------------------------------------|
| int  | interface = 0 | Get the first <b>valid</b> interface. |

### Returns:

DHCP\_NOTNEEDED --- We had a properly configured IP Address

DHCP\_OK --- We got a valid IP Address

DHCP\_FAILED --- We needed an IP Address, but did not get one

## 5.10. GetDHCPState

### Synopsis:

```
int GetDHCPState( );
```

### Description:

This function gets the current DHCP state.

### Parameters:

None

### Returns:

SDHCP\_DISCOVER --- The system is discovering the DHCP servers  
SDHCP\_OFFER --- The system has responded to an Offer  
SDHCP\_ACK --- The System has acknowledged the Offer  
SDHCP\_INIT --- The System is reinitializing  
SDHCP\_CMPL --- The System has completed the last transaction  
SDHCP\_RENEW --- The System is in the process of renewing  
SDHCP\_REBIND --- The System has failed the Renew and is trying to Rebind  
SDHCP\_RELEASE --- The System is trying to release the Lease  
SDHCP\_NOTSTARTED --- The System has not been initialized



## 6. FTP Client Library

### 6.1. Introduction

The FTP Client module provides code for sending and receiving files from a FTP server. Most embedded platforms, including the NetBurner embedded development environment, do not have a built-in file system (although you could implement a file system on your own). This means that the Client FTP code does not use files as you might think of them on a computer's hard drive. Instead, you will create data streams that are associated with file names. In other words, a "file" is just a collection of bytes that can be stored in Flash memory or RAM.

When you read a file from the external FTP server using the FTPGetFile function, you will receive a file descriptor (fd), not a file. If you read the bytes/data from this file descriptor you will receive the contents of the remote file as a stream of data. The received data is stored in Flash or RAM.

When you create a file on a remote FTP server using the FTPSendFile, function, then you write the stream of data that will become the remote file associated with the file descriptor. If the remote FTP server has a file system (e.g. a Unix server running a FTP daemon), then the stream of data you wrote would likely be stored as a file on a computer's hard drive.

Some basic knowledge of the inner workings of FTP will be helpful in using this module. Two recommended references are TCP/IP Illustrated Volume 1 (Chapter 27) by Richard Stevens, and/or RFC 959.

### Required Header File

```
#include <ftp.h>          // Found in C:\Nburn\include
```

### FTP Client Module Description

The basic procedure to use the FTP Client module is:

1. Open the FTP session (with the FTP\_InitializeSession function)
2. Send commands within the FTP session (with the FTPGetList, FTPGetFileNames, FTPGetFile, and/or FTPSendFile functions)
3. Close the session (with the FTP\_CloseSession function)

### FTP Client Functions to Initialize and/or Close a FTP Session

- FTP\_InitializeSession --- Create/Initialize a connection to an FTP Server
- FTP\_CloseSession --- Close the FTP session

### FTP Client Directory Functions

- FTPGetDir --- Get the current working directory
- FTPSetDir --- Set the current working directory
- FTPDeleteDir --- Delete a directory

- FTPMakeDir --- Make a directory
- FTPUpDir --- Move up one directory level

## FTP Client Miscellaneous File Functions

- FTPDeleteFile --- Delete a file on the server
- FTPRenameFile --- Rename a file on the server

## FTP Client Send File Function

- FTPSendFile --- Setup to send a file on an existing FTP session

## FTP Client Get File Functions

- FTPGetFile --- Setup to receive a file on an existing FTP session
- FTPGetList --- Setup to receive a directory on an existing FTP session
- FTPGetFileNames --- Setup to receive a just the file names from the existing FTP session

## FTP Client Low Level Functions

- FTPRawCommand --- Send a command and get a response from the control connection
- FTPGetCommandResult --- Get a response from the control connection without sending a command
- FTPRawStreamCommand --- Send a command and get a response over a stream connection

## FTP Client Example Program

```
#include <predef.h>
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ftp.h>
extern "C" {
void UserMain(void * pd);
}
#define SERVER_IP_ADDR "10.1.1.3"
#define FTP_PORT 21
#define USER "username"
#define PASS "password"
char tmp_resultbuff[255];
// This function reads the data stream from the fd and
// displays it to stdout
void ShowFileContents(int fdr)
{
    iprintf("\r\n[");
    int rv;
```

```

do
{
rv = ReadWithTimeout(fdr,tmp_resultbuff,255,20);
if (rv < 0) fprintf("RV=%d\r\n",rv);
else
{
tmp_resultbuff[rv] = 0;
fprintf("%s",tmp_resultbuff);
}
}
while (rv > 0);
fprintf("]\r\n");
}
void UserMain(void * pd)
{
InitializeStack(); // Initialize TCP/IP Stack
EnableAutoUpdate(); // Enable NetBurner AutoUpdate downloads
OSChangePrio(MAIN_PRIO); // Set UserMain priority level to default
int ftp =
FTP_InitializeSession(AsciiToIp(SERVER_IP_ADDR),FTP_PORT,USER,PASS,100)
;
if (ftp > 0)
{
int rv = 0;
// Change to the test directory
//*****WARNING*****
// To run this sample, a test1 dir must exist on the test server
//*****WARNING*****
rv = FTPSetDir(ftp,"test1",100);
if (rv == FTP_OK)
{
fprintf("Getting the file names from this directory\r\n");
int fdr = FTPGetList(ftp,NULL,100);
if (fdr > 0)
{
ShowFileContents(fdr);
close(fdr);
// Read the command result code from GetFilename command
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
fprintf("Error Command result = %d
%s\r\n",rv,tmp_resultbuff);
}
The NetBurner Runtime Libraries User Manual NetBurner, Inc.
else
fprintf("Failed to get file list\r\n");
fprintf("Now creating the sample file FOOBAR.TXT\r\n");
fdr = FTPSendFile(ftp,"FOOBAR.TXT",FALSE,100);
if (fdr > 0)
{
writestring(fdr,"This is a test file\r\n");
writestring(fdr,"This is line 2 of the test file\r\n");
writestring(fdr,"Last Line\r\n");
close(fdr);
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
fprintf("Error Command result = %d

```

```
%s\r\n",rv,tmp_resultbuff);
iprintf("Now trying to read back the file we created \r\n");
fdr = FTPGetFile(ftp,"FOOBAR.TXT",FALSE,100);
if (fdr > 0)
{
ShowFileContents(fdr);
close(fdr);
// Read the command result code from the GetFilename
command
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
iprintf("Error Command result = %d
%s\r\n",rv,tmp_resultbuff);
}
else
iprintf("Failed to get file FOOBAR.TXT\r\n");
}
else
iprintf("Failed to create file FOOBAR.TXT\r\n");
}
else
iprintf("Failed to change to test directory");
FTP_CloseSession(ftp);
}
else
iprintf("Failed to open FTP Session\r\n");
while(1);
}
```

## 6.2. FTP\_InitializeSession

### Synopsis:

```
int FTP_InitializeSession( IPADDR server_address, WORD port, PCSTR
UserName, PCSTR PassWord, DWORD time_out );
```

### Description:

This function creates and initializes a connection to an FTP server. This call creates a connection to a FTP server, and logs in with the username and password specified in the function call. The session handle returned from this call is used by the FTP file and directory functions. **Note:** The FTP\_CloseSession function is used to close the session.

### Parameters:

| Type   | Name           | Description                                  |
|--------|----------------|--|
| IPADDR | server_address | The IP Address of the FTP Server.            |
| WORD   | port           | The port number to connect to on the Server. |
| PCSTR  | UserName       | The account User Name.                       |
| PCSTR  | PassWord       | The account password.                        |
| DWORD  | time_out       | The number of time ticks to wait.            |

### Return Values:

> 0 --- FTP session handle  
FTP\_TIMEOUT --- Time out  
FTP\_PASSWORDERROR --- Password error  
FTP\_CONNECTFAIL --- Network error  
FTP\_NETWORKERROR --- Network error

## 6.3. FTP\_CloseSession

### Synopsis:

```
int FTP_CloseSession( int session );
```

### Description:

This function closes the specified FTP session. This function should be called when an FTP session is complete.

**Warning:** Failure to call this function will result in memory/resource leaks.

### Parameter:

| Type | Name    | Description             |
|------|---------|-------------------------|
| int  | session | The FTP session handle. |

### Return Values:

FTP\_OK --- Closed successfully  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command error  
FTP\_NETWORKERROR --- Network error  
FTP\_BADSESSION --- Invalid session number

## 6.4. FTPGetDir

### Synopsis:

```
int FTPGetDir( int ftp_Session, char * dir_buf, int nbytes, WORD
timeout );
```

### Description:

This function gets the current working directory name on the FTP server. This function also copies the name of the current working directory into the buffer specified by dir\_buf.

### Parameters:

| Type | Name        | Description  |
|------|-------------|--|
| int  | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call. |
| char | *dir_buf    | A pointer to the buffer that will hold the directory name.           |
| int  | nbytes      | The maximum number of bytes to copy.                                 |
| WORD | timeout     | The number of ticks to wait for timeout.                             |

### Return Values:

> 0 --- The number of bytes read  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Could not execute command  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.5. FTPSetDir

### Synopsis:

```
int FTPSetDir( int ftp_Session, const char * new_dir, WORD timeout );
```

### Description:

This function sets the current working directory of the FTP Server.

### Parameters:

| Type       | Name        | Description  |
|------------|-------------|--|
| int        | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *new_dir    | The name of the directory to change to                               |
| WORD       | timeout     | The number of timer ticks to wait for timeout.                       |

### Return Values:

FTP\_OK --- Set successfully

FTP\_TIMEOUT --- Time out

FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)

FTP\_CONNECTFAIL --- FTP failure

FTP\_NETWORKERROR --- Network error



## 6.6. FTPDeleteDir

### Synopsis:

```
int FTPDeleteDir( int ftp_Session, const char * dir_to_delete, WORD
timeout );
```

### Description:

This function deletes a directory on the FTP server.

### Parameters:

| Type       | Name           | Description  |
|------------|----------------|--|
| int        | ftp_Session    | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *dir_to_delete | The name of the directory to be deleted.                             |
| WORD       | timeout        | The number of ticks to wait for a timeout.                           |

### Return Values:

FTP\_OK --- Deleted successfully

FTP\_TIMEOUT --- Time out

FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)

FTP\_CONNECTFAIL --- FTP failure

FTP\_NETWORKERROR --- Network error

## 6.7. FTPMakeDir

### Synopsis:

```
int FTPMakeDir( int ftp_Session, const char * dir_to_make, WORD timeout
);
```

### Description:

This function makes a directory on the FTP server.

### Parameters:

| Type       | Name         | Description  |
|------------|--------------|--|
| int        | ftp_Session  | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *dir_to_make | The name of the directory to create.                                 |
| WORD       | timeout      | The number of ticks to wait for a timeout.                           |

### Return Values:

FTP\_OK --- Created successfully  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.8. FTPUpDir

### Synopsis:

```
int FTPUpDir( int ftp_Session, WORD timeout );
```

### Description:

This function moves up one directory level in the directory hierarchy on the FTP server.

### Parameters:

| Type | Name        | Description  |
|------|-------------|--|
| int  | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call. |
| WORD | timeout     | The number of ticks to wait for a timeout.                           |

### Return Values:

FTP\_OK --- Changed directory successfully  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.9. FTPDeleteFile

### Synopsis:

```
int FTPDeleteFile( int ftp_Session, const char * file_name, WORD
timeout );
```

### Description:

This function deletes a file on the FTP server.

### Parameters:

| Type       | Name        | Description  |
|------------|-------------|--|
| int        | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *file       | The file name to be deleted.   |
| WORD       | timeout     | The number of ticks to wait for a timeout.                           |

### Return Values:

FTP\_OK --- Deleted successfully  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.10. FTPRenameFile

### Synopsis:

```
int FTPRenameFile( int ftp_Session, const char * old_file_name, const
char * new_file_name, WORD timeout );
```

### Description:

This function renames a file on the FTP server.

### Parameters:

| Type       | Name           | Description  |
|------------|----------------|--|
| int        | ftp_Session    | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *old_file_name | The file name to be renamed.   |
| const char | *new_file_name | The new file name.   |
| WORD       | timeout        | The number of ticks to wait for a timeout.                           |

### Return Values:

FTP\_OK --- Renamed successfully  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.11. FTPSendFile

### Synopsis:

```
int FTPSendFile( int ftp_Session, const char * full_file_name, BOOL
bBinaryMode, WORD timeout );
```

### Description:

This function call initializes the send file process to send a file to a FTP server. It sets up a new TCP connection that will be used to transfer the file data. To actually send the file data, use the returned file descriptor and the standard I/O write commands, such as write, writestring, or writeall. After sending the data, close the returned file descriptor with the close function.

**Important:** After the file has been sent, you must call FTPGetCommandResult to get the result from the write. **Warning:** Failing to do this will cause the system to get out of sync. A return value of 226 is normal.

### Parameters:

| Type       | Name            | Description  |
|------------|-----------------|--|
| int        | ftp_Session     | The FTP session handle returned from the FTP_InitializeSession call.   |
| const char | *full_file_name | The file name that the FTP server should assign to the data sent.      |
| BOOL       | bBinaryMode     | True if the file is to be transferred in binary mode. False for ASCII. |
| WORD       | timeout         | The number of ticks to wait for a timeout.                             |

### Return Values:

> 0 --- FTP write file descriptor  
 FTP\_TIMEOUT --- Time out  
 FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
 FTP\_CONNECTFAIL --- FTP failure  
 FTP\_NETWORKERROR --- Network error

### Example:

```
// Setup to send file and get file descriptor fd
// The "ftp" session handle would has already been created by the
// FTP_Initialize Session() call
int fd = FTPSendFile(ftp,"FOOBAR.TXT",FALSE,100);
if (fd > 0)
{
    writestring(fd,"This is a test file\r\n");
    writestring(fd,"This is line 2 of the test file\r\n");
    writestring(fd,"Last Line\r\n");
}
```

```
close(fd);
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
    iprintf("Write Error Command result = %d
    %s\r\n",rv,tmp_resultbuff);
else
    iprintf("Failed to create file FOOBAR.TXT\r\n");
```

## 6.12. FTPGetFile

### Synopsis:

```
int FTPGetFile( int ftp_Session, const char * full_file_name, BOOL
bBinaryMode, WORD timeout );
```

### Description:

This function call initializes the receive file process used to get a file from a FTP server. It sets up a new TCP connection that will be used to transfer the file data. To actually receive the file data, use the returned file descriptor and the standard I/O read commands, such as the ReadWithTimeout function.

**Note:** It would be unwise to use the read function, because it would block forever if the connection were lost to the FTP Server. After reading the data, close the returned file descriptor with the close function.

**Important:** After the file descriptor has been returned, you must call the FTPGetCommandResult function to get the result from the read. **Warning:** Failing to do this will cause the system to get out of sync. A return value of 226 is normal.

### Parameters:

| Type       | Name            | Description  |
|------------|-----------------|--|
| int        | ftp_Session     | The FTP session handle returned from the FTP_InitializeSession call.   |
| const char | *full_file_name | The complete file name to be transferred, including path.              |
| BOOL       | bBinaryMode     | True if the file is to be transferred in binary mode. False for ASCII. |
| WORD       | timeout         | The number of ticks to wait for a timeout.                             |

### Return Values:

> 0 --- FTP read file descriptor  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

### Example:

```
// Setup file transfer and get file descriptor fdr
int fdr = FTPGetFile(ftp,"FOOBAR.TXT",FALSE,100);
if (fdr > 0)
{
// The following function reads data from the specified file until
complete.
```



```
// This is the location where you could use
// a different mechanism to retrieve the data.
ShowFileContents(fdr);
close(fdr);
// Now read the command result code from the GetFile command
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
    fprintf("Error Command result = %d %s\r\n",rv,tmp_resultbuff);
}
else
    fprintf("Failed to get file FOOBAR.TXT\r\n");
// This function reads the data stream from the fd and
// displays it to stdout, which is usually the com1 serial
// port on the NetBurner board.
void ShowFileContents(int fdr)
{
    fprintf("\r\n[");
    int rv;
    do
    {
        rv = ReadWithTimeout(fdr,tmp_resultbuff,255,20);
        if (rv < 0) fprintf("RV = %d\r\n",rv);
        else
        {
            tmp_resultbuff[rv] = 0;
            fprintf("%s",tmp_resultbuff);
        }
    }
    while (rv > 0);
    fprintf("]\r\n");
}
```

## 6.13. FTPGetList

### Synopsis:

```
int FTPGetList( int ftp_Session, const char * full_dir_name, WORD
timeout );
```

### Description:

This function initializes the get directory process to receive a full directory listing from the FTP server. A new TCP connection is created to receive the file listing from the server. To actually receive the list use the returned file descriptor to read the list using the standard I/O read commands, such as ReadWithTimeout. After reading the data, close the returned file descriptor with the close function.

**Important:** After you have received the list, you must call the FTPGetCommandResult function to get the result from the read. **Warning:** Failing to do this will cause the system to get out of sync. A return value of 226 is normal.

### Parameters:

| Type       | Name           | Description  |
|------------|----------------|--|
| int        | ftp_Session    | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *full_dir_name | The complete directory name to be transferred. Can be NULL.          |
| WORD       | timeout        | The number of ticks to wait for a timeout.                           |

### Return Values:

> 0 --- FTP read file descriptor  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

### Example:

```
int fdr = FTPGetList(ftp,NULL,100);
if (fdr > 0)
{
//This function reads data from the fd until complete.
ShowFileContents(fdr);
// You would probably use a different function.
// The source for this function is shown in the module example.
close(fdr);
// Now read the command result code from the GetList command
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
```

```
iprintf("Error Command result = %d %s\r\n",rv,tmp_resultbuff);
}
else
iprintf("Failed to get file list\r\n");
// This function reads the data stream from the fd and displays
// it to stdout, which is usually the com1 serial port on the
// NetBurner board.
void ShowFileContents(int fdr)
{
iprintf("\r\n[");
int rv;
do
{
rv = ReadWithTimeout(fdr,tmp_resultbuff,255,20);
if (rv < 0) iprintf("RV = %d\r\n",rv);
else
{
tmp_resultbuff[rv] = 0;
iprintf("%s",tmp_resultbuff);
}
}
while (rv > 0);
iprintf("]\r\n");
}
```

## 6.14. FTPGetFileNames

### Synopsis:

```
int FTPGetFileNames( int ftp_Session, const char * full_dir_name, WORD
timeout );
```

### Description:

This function initializes the get directory process to receive just the file names listing from the server. It sets up a new TCP connection to receive the file listing from the server. To actually receive the list, use the returned file descriptor to read the data using the standard I/O read commands, such as ReadWithTimeout.

**Warning:** It would be unwise to use the read function, because it would block forever if the connection were lost to the FTP Server. After reading the data, close the returned file descriptor with the close function.

After the file connection has been established, you **must** call the FTPGetCommandResult function to get the result from the read. **Warning:** Failing to do this will cause the system to get out of sync. A return value of 226 is normal.

### Parameters:

| Type       | Name           | Description  |
|------------|----------------|--|
| int        | ftp_Session    | The FTP session handle returned from the FTP_InitializeSession call. |
| const char | *full_dir_name | The complete file name to be transferred, including the path.        |
| WORD       | timeout        | The number of ticks to wait for a timeout.                           |

### Return Values:

```
int --- The command success code
> 0 --- FTP read file descriptor
FTP_TIMEOUT --- Time out
FTP_CONNECTFAIL --- FTP failure
FTP_NETWORKERROR --- Network error
```

### Example:

```
int fdr = FTPFileNames(ftp,NULL,100);
if (fdr > 0)
{
// This function reads data from the fd until complete. You may
// want to use a different method here.
ShowFileContents(fdr);
close(fdr);
// Now read the command result code from the GetList command
```

```
rv = FTPGetCommandResult(ftp,tmp_resultbuff,255,100);
if (rv != 226)
iprintf("Error Command result = %d %s\r\n",rv,tmp_resultbuff);
}
else
iprintf("Failed to get file list\r\n");
// This function reads the data stream from the fd and
// displays it to stdout, which is usually the com1 serial
// port on the NetBurner board.
void ShowFileContents(int fdr)
{
iprintf("\r\n[");
int rv;
do
{
rv = ReadWithTimeout(fdr,tmp_resultbuff,255,20);
if (rv < 0) iprintf("RV = %d\r\n",rv);
else
{
tmp_resultbuff[rv] = 0;
iprintf("%s",tmp_resultbuff);
}
}
while (rv > 0);
iprintf("]\r\n");
}
```

## 6.15. FTPRawCommand

### Synopsis:

```
int FTPRawCommand( int ftp_Session, const char * cmd, char * cmd_buf,  
int nbytes, WORD timeout );
```

### Description:

This function sends a command and gets a response from the FTP control connection. This command is the basis for most of the FTP commands in the FTP module. It is used where a stream return is not expected. **Important:** Using this command **requires** that you are **familiar** with the **FTP Protocol** details.

### Parameters:

| Type       | Name        | Description  |
|------------|-------------|--|
| int        | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call.                                 |
| const char | *cmd        | The command to send - <b>not</b> including the \r\n termination.                                     |
| char       | *cmd_buf    | The buffer to hold the result from the server which includes a CLF/LF at the end.                    |
| int        | nbytes      | The maximum number of bytes in server response to copy (including numeric code and null terminator). |
| WORD       | timeout     | The number of ticks to wait for a timeout.   |

### Return Values:

> 0 --- The FTP numeric response code  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.16. FTPGetCommandResult

### Synopsis:

```
int FTPGetCommandResult( int ftp_Session, char * cmd_buf, int nbytes,  
WORD timeout );
```

### Description:

This function gets a response from the control connection without sending a command. This command may be used after the following four functions to get the command result:

1. FTPGetList
2. FTPGetFileNames
3. FTPGetFile
4. FTPSendFile

### Parameters:

| Type | Name        | Description  |
|------|-------------|--|
| int  | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call. |
| char | *cmd_buf    | The buffer to hold the result.                                       |
| int  | nbytes      | The maximum number of bytes to copy.                                 |
| WORD | timeout     | The number of ticks to wait for a timeout.                           |

### Return Values:

> 0 --- The FTP read file descriptor  
FTP\_TIMEOUT --- Time out  
FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
FTP\_CONNECTFAIL --- FTP failure  
FTP\_NETWORKERROR --- Network error

## 6.17. FTPRawStreamCommand

### Synopsis:

```
int FTPRawStreamCommand( int ftp_Session, const char * cmd, int
* pResult, char * cmd_buf, int nbytes, WORD timeout );
```

### Description:

This function sends a command and gets a response over a stream connection. This command is the basis for functions such as FTPGetList and FTPGetFiles. It is used where a stream return is expected.

After you have received or sent the data stream you **must** call FTPGetCommandResult to get the result from the read or write. **Warning: Failing to do this will cause the system to get out of sync.** A return value of 226 is normal. However, other values such as 250 are also possible depending on the FTP server.

### Parameters:

| Type       | Name        | Description  |
|------------|-------------|--|
| int        | ftp_Session | The FTP session handle returned from the FTP_InitializeSession call.                                       |
| const char | *cmd        | The command to send - <b>not</b> including the \r\n termination.   |
| int        | *pResult    | The FTP command result code.   |
| char       | *cmd_buf    | The buffer to hold the command connection results from the server, which includes a CR/LF at the end.      |
| int        | nbytes      | The maximum number of bytes to copy into the cmd_buf (includes numeric response code and null terminator). |
| WORD       | timeout     | The number of ticks to wait for a timeout.   |

### Return Values:

> 0 --- FTP data channel file descriptor (Note: The FTP server will drop the data channel after completion of sending data to the client that will cause a read error)  
 FTP\_TIMEOUT --- Time out  
 FTP\_COMMANDFAIL --- Command failed (e.g. a nonexistent directory)  
 FTP\_CONNECTFAIL --- FTP failure  
 FTP\_NETWORKERROR --- Network error



## 7. FTP Server Library

### 7.1. Introduction

Implementing an FTP server in an embedded system without a built-in file system is not a trivial undertaking. Most embedded applications do not require a file system, and a file system is not part of the standard NetBurner package. If a file system is required for a specific application, it is the responsibility of the programmer to implement the required features. A file system could be the trivially simple example of a single file, or it could be quite complex.

Using the FTP Server requires that you, the programmer, write the functions defined in the FTP documentation. These functions are "callback" functions that allow you to customize the FTP server actions to suit your particular application. **Note:** All callback functions **required** for your application **must** be implemented by you.

The NetBurner examples (C:\Nburn\examples) directory has three FTP server sample applications:

1. ftpd\_trivial --- A simple example that reads and writes a single file.
2. ftpd\_expose\_html --- A more complex example that exposes all HTML served files to the FTP server.
3. ftpd\_code\_update --- This example shows you how to upgrade the NetBurner firmware and reset the system using FTP.

### Required Header File

```
#include <ftpd.h>          // Found by default in C:\Nburn\include
```

### Operational Functions

- FTPDStart --- Starts the FTP Server task
- FTPDStopReq --- Sends a stop request to the currently running FTPD

### FTP Session Callback typedef

- (FTPDCallBackReportFunct) --- The typedef for all directory reporting callbacks

### FTP Session Callback Functions (These functions must be implemented by the programmer)

- FTPDSessionStart --- Function called to indicate the start of a user session
- FTPDSessionEnd --- Function called to indicate a user session will be terminated

**FTP Directory Callback Functions (These functions must be implemented by the programmer)**

- `FTPD_DirectoryExists` --- Function called by the FTP Server to test for the existence of a directory
- `FTPD_CreateSubDirectory` --- Function called by the FTP Server to create a directory
- `FTPD_DeleteSubDirectory` --- Function called by the FTP Server to delete a directory
- `FTPD_ListSubDirectories` --- Function called by the FTP Server to list all subdirectories under the current directory

**FTP File Callback Functions (These functions must be implemented by the programmer)**

- `FTPD_FileExists` --- Function to report on the whether or not a file exists
- `FTPD_SendFileToClient` --- Function to send the contents of a file to a file descriptor
- `FTPD_AbleToCreateFile` --- Function to report on the ability to create/receive a file
- `FTPD_GetFileFromClient` --- Function to create/get a file
- `FTPD_DeleteFile` --- Function to delete a file
- `FTPD_DeleteFile` --- User supplied function to delete a file
- `FTPD_ListFile` --- Lists every file in the current directory
- `FTPD_ListFile` --- User supplied function that lists every file in the directory
- `FTPD_Rename` --- User supplied function to rename a file

## 7.2. FTPDStart

### Synopsis:

```
int FTPDStart( WORD port, BYTE server_priority );
```

### Description:

This function starts the FTP Server task, which listens for incoming connections.

**Warning:** Only one instance of the FTPD is allowed.

### Parameters:

| Type | Name            | Description  |
|------|-----------------|--|
| WORD | port            | The TCP port to listen to for incoming FTP requests. |
| BYTE | server_priority | The uC/OS task priority for the FTP Server.          |

### Return Values:

FTPD\_RUNNING --- The FTPD is already running  
FTPD\_LISTEN\_ERR --- The listen socket could not be opened  
FTPD\_OK --- The FTPD was successfully started  
FTPD\_FAIL --- The FTPD task could not be created

## 7.3. FTPDStopReq

### Synopsis:

```
int FTPDStopReq( );
```

### Description:

This function sends a stop request to the currently running FTPD.

### Parameters:

None

### Return Values:

FTPD\_RUNNING --- The FTPD is still running

FTPD\_NOT\_RUNNING --- The FTPD is no longer running

## 7.4. ( FTPDCallBackReportFunct )

### Synopsis:

```
typedef void ( FTPDCallBackReportFunct )( int handle, const char  
* name_to_report );
```

### Description:

This is the typedef for all directory reporting callbacks. This callback type definition is used by the directory reporting functions.

### Parameters:

| Type       | Name            | Description   |
|------------|-----------------|---|
| int        | handle          | The handle passed into the listing function.            |
| const char | *name_to_report | The file name to report for use in a directory listing. |

### Return Value:

Nothing --- This is a void function

## 7.5. FTPDSessionStart

### Synopsis:

```
void * FTPDSessionStart( const char  * user, const char  * passwd,  
const IPADDR hi_ip );
```

### Description:

This function is called to indicate the start of a user Session. This function is called following the creation of a **new** FTP session. This function needs to determine the validity of the user/password pair. The returned void pointer **will** be passed to **all** access functions, which will then be asked to determine the validity of the operation based on the permissions associated with the return value.

### Parameters:

| Type       | Name         | Description   |
|------------|--------------|---|
| const char | *user        | The name of the user attempting to establish an FTP session.      |
| const char | *passwd      | The password of the user attempting to establish an FTP session.  |
| const      | IPADDR hi_ip | The IP Address of the Server trying to establish this connection. |

### Return Values:

NULL --- The user name/password set is invalid

(obj) --- A non-null void pointer to an object that will be associated with this login session

## 7.6. FTPDSessionEnd

### Synopsis:

```
void FTPDSessionEnd( void * pSession );
```

### Description:

This function is called to indicate that a user session will be terminated. This callback function gives the user program the opportunity to clean up any storage associated with the void pointer returned from the FTPDSessionStart call.

### Parameter:

| Type | Name      | Description   |
|------|-----------|---|
| void | *pSession | The void * object returned from the FTPDSessionStart function call. |

### Return Value:

Nothing --- This is a void function

## 7.7. FTPD\_DirectoryExists (User Defined)

### Synopsis:

```
int FTPD_DirectoryExists( const char * full_directory, void * pSession
);
```

### Description:

This function called by the FTP Server to test for the existence of a directory. This function is called by the FTP Server as the result of an attempt to change to a new directory. This function can also be used to validate the permissions of the session. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name            | Description   |
|------------|-----------------|---|
| const char | *full_directory | The name of the new directory to test.                              |
| void       | *pSession       | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested directory exists

FTP\_FAIL --- The requested directory does not exist, or access is not permitted for the user



## 7.8. FTPD\_CreateSubDirectory (User Defined)

### Synopsis:

```
int FTPD_CreateSubDirectory( const char * current_directory, const char
* new_dir, void * pSession );
```

### Description:

This function is called by the FTP Server to create a directory. This function is called by the FTP Server as the result of an attempt to create a new directory. This function can also be used to validate the permissions of the session. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name               | Description   |
|------------|--------------------|---|
| const char | *current_directory | The current value of the session directory.                         |
| const char | *new_dir           | The directory to create under the current_directory.                |
| void       | *pSession          | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested directory was created  
FTP\_FAIL --- The requested directory could not be created

## 7.9. FTPD\_DeleteSubDirectory (User Defined)

### Synopsis:

```
int FTPD_DeleteSubDirectory( const char  *current_directory, const char
* sub_dir, void * pSession );
```

### Description:

This function is called by the FTP Server to delete a directory. This function is called by the FTP Server as the result of an attempt to delete a subdirectory. This function call can be used to validate the permissions of this session. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name               | Description   |
|------------|--------------------|---|
| const char | *current_directory | The current value of the session directory.                         |
| const char | *sub_dir           | The directory to delete under the current_directory                 |
| void       | *pSession          | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested directory was deleted

FTP\_FAIL --- The requested directory could not be deleted

## 7.10. FTPD\_ListSubDirectories (User Defined)

### Synopsis:

```
int FTPD_ListSubDirectories( const char * current_directory, void
* pSession, FTPDCallBackReportFunct * pFunc, int handle );
```

### Description:

This function is called by the FTP Server to list all subdirectories under the current directory. This function is called by the FTP Server as the result of a client's attempt to list the contents of a directory. **This function must be implemented by the programmer.**

### Parameters:

| Type                    | Name               | Description  |
|-------------------------|--------------------|--|
| const char              | *current_directory | The current value of the session directory.                              |
| void                    | *pSession          | The void * object returned from the FTPDSessionStart function call.      |
| FTPDCallBackReportFunct | *pFunc             | The pointer to the callback function to be called for each subdirectory. |
| int                     | handle             | The handle value to be passed back into the pFunc.                       |

### Return Values:

FTP\_OK --- The requested listing was successfully delivered

FTP\_FAIL --- The requested directory could not be listed

### Example:

**Everything inside the callback function stub must be supplied by the programmer.**

The FTP server will automatically call this function and provide values for the function variables. It is the programmer's responsibility to execute pFunc( ) with the provided handle and a pointer to the string representing the subdirectory name. **Note: pFunc( ) must be executed once for each subdirectory name.** In the example below, the variables number\_of\_directories and DirectoryName **must** be declared **and** initialized elsewhere in the application program:

```
int FTPD_ListSubDirectories(const char *current_directory, void
*pSession, FTPDCallBackReportFunct *pFunc, int handle);
{
    for (int n = 0; n < number_of_dir; n++)
        pFunc(handle, DirectoryName[n]);
    return FTP_OK;
}
```

## 7.11. FTPD\_FileExists (User Defined)

### Synopsis:

```
int FTPD_FileExists( const char * full_directory, const char
* file_name, void * pSession );
```

### Description:

This function reports on whether or not a file exists. This function checks for the existence of a file, usually just before an attempt is made to download the file. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name            | Description   |
|------------|-----------------|---|
| const char | *full_directory | The current value of the session directory.                         |
| const char | *file_name      | The name of the file to check.                                      |
| void       | *pSession       | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested file exists  
FTP\_FAIL --- The requested file does not exist

## 7.12. FTPD\_SendFileToClient (User Defined)

### Synopsis:

```
int FTPD_SendFileToClient( const char * full_directory, const char
* file_name, void * pSession, int fd );
```

### Description:

This function sends the contents of a file to a file descriptor. This function sends a file to an FTP client. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name            | Description   |
|------------|-----------------|---|
| const char | *full_directory | The current value of the session directory.                         |
| const char | *file_name      | The name of the file to send.                                       |
| void       | *pSession       | The void * object returned from the FTPDSessionStart function call. |
| int        | fd              | The file descriptor to send to.                                     |

### Return Values:

FTP\_OK --- The requested file was sent

FTP\_FAIL --- The requested file was not sent

## 7.13. FTPD\_AbleToCreateFile (User Defined)

### Synopsis:

```
int FTPD_AbleToCreateFile( const char * full_directory, const char
* file_name, void * pSession );
```

### Description:

This function will report on the ability to create/receive a file. This function determines if a file can be created. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name            | Description   |
|------------|-----------------|---|
| const char | *full_directory | The current value of the session directory.                         |
| const char | *file_name      | The name of the file to create.                                     |
| void       | *pSession       | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested file can be written (i.e. created)

FTP\_FAIL --- The requested file could not be created

## 7.14. FTPD\_GetFileFromClient (User Defined)

### Synopsis:

```
int FTPD_GetFileFromClient( const char * full_directory, const char
* file_name, void * pSession, int fd );
```

### Description:

This function is used to create/get a file or to receive a file from the FTP client. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name            | Description   |
|------------|-----------------|---|
| const char | *full_directory | The current value of the session directory.                         |
| const char | *file_name      | The name of the file to create.                                     |
| void       | *pSession       | The void * object returned from the FTPDSessionStart function call. |
| int        | fd              | The file descriptor that will be used to receive the file.          |

### Return Values:

FTP\_OK --- The requested file was written (i.e. created)  
FTP\_FAIL --- The requested file was not created

## 7.15. FTPD\_DeleteFile

### Synopsis:

```
int FTPD_DeleteFile( const char * current_directory, const char
* file_name, void * pSession );
```

### Description:

This function is used to delete a file.

### Parameters:

| Type       | Name               | Description   |
|------------|--------------------|---|
| const char | *current_directory | The current value of the session directory.                         |
| const char | *file_name         | The name of the file to delete.                                     |
| void       | *pSession          | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested file was deleted

FTP\_FAIL --- The requested file was not deleted



## 7.16. FTPD\_DeleteFile (User Defined)

### Synopsis:

```
int FTPD_DeleteFile( const char * current_directory, const char
* file_name, void * pSession );
```

### Description:

This is a user written function to delete a file. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name               | Description   |
|------------|--------------------|---|
| const char | *current_directory | The current value of the session directory.                         |
| const char | *file_name         | The name of the file to delete.                                     |
| void       | *pSession          | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested file was deleted  
FTP\_FAIL --- The requested file could not be deleted

## 7.17. FTPD\_ListFile

### Synopsis:

```
int FTPD_ListFile( const char * current_directory, void * pSession,  
FTPCallBackReportFunct * pFunc, int handle );
```

### Description:

This is a callback function, with the name of every file in the directory.

### Parameters:

| Type                   | Name               | Description   |
|------------------------|--------------------|---|
| const char             | *current_directory | The current value of the session directory.   |
| void                   | *pSession          | The void * object returned from the FTPDSessionStart function call.   |
| FTPCallBackReportFunct | *pFunc             | The pointer to the callback function to be called for each file name. This is a callback function provided and used by the NetBurner internal FTP code. |
| int                    | handle             | The handle value to be passed back into the pFunc. This is a handle provided and used by the NetBurner internal FTP code.                               |

### Return Values:

FTP\_OK --- The requested files were listed

FTP\_FAIL --- The requested files were not listed

## 7.18. FTPD\_ListFile (User Defined)

### Synopsis:

```
int FTPD_ListFile( const char * current_directory, void * pSession,
FTPDCallBackReportFunct * pFunc, int handle );
```

### Description:

This function is a user supplied function that lists all files in the current directory. **This function must be implemented by the programmer.**

### Parameters:

| Type                    | Name               | Description   |
|-------------------------|--------------------|---|
| const char              | *current_directory | The current value of the session directory.   |
| void                    | *pSession          | The void * object returned from the FTPDSessionStart function call.   |
| FTPDCallBackReportFunct | *pFunc             | The pointer to the callback function to be called for each file name. This is a callback function provided and used by the NetBurner internal FTP code. |
| int                     | handle             | The handle value to be passed back into the pFunc. This is a handle provided and used by the NetBurner internal FTP code.                               |

### Return Values:

FTP\_OK --- The requested files were listed  
FTP\_FAIL --- The requested files were not listed

### Example:

**Everything** inside the callback function stub **must** be supplied by the programmer. The FTP server **will** automatically **call** this function **and** provide values for the function variables.

**Important:** It is the **programmer's responsibility** to execute pFunc( ) with the provided handle and a pointer to the string representing the file name.

**Note:** pFunc( ) **must** be executed **once** for **each** file name. In the example on the next page, the variables number\_of\_directories and FileNames **must** be declared **and** initialized elsewhere in the application program:

```
int FTPD_ListFile( const char *current_directory, void *pSession,
FTPDCallBackReportFunct *pFunc, int handle );
{
    for (int n = 0; n < numberof_files; n++)
        pFunc(handle, FileNames[n]);
    return FTP_OK;
}
```

## 7.19. FTPD\_Rename (User Defined)

### Synopsis:

```
int FTPD_Rename( const char * current_directory, const char
* cur_file_name, const char * new_file_name, void * pSession );
```

### Description:

This is a user written function; this function is used to rename a file. **This function must be implemented by the programmer.**

### Parameters:

| Type       | Name               | Description   |
|------------|--------------------|---|
| const char | *current_directory | The current value of the session directory.                         |
| const char | *cur_file_name     | The current name of the file to rename.                             |
| const char | *new_file_name     | The new file name.  |
| void       | *pSession          | The void * object returned from the FTPDSessionStart function call. |

### Return Values:

FTP\_OK --- The requested file was deleted

FTP\_FAIL --- The requested file could not be renamed

## 8. HTTP and HTML Libraries

### Required Header Files

```
#include <http.h>           // Found in C:\Nburn\include
#include <htmlfiles.h>       // Found in C:\Nburn\include
```

### HTTP Dameon Functions

- StartHTTP --- Starts the HTTP server
- StopHTTP --- Stops the HTTP subsystem
- SetNewPostHandler --- Setup a custom Post Handler
- SetNewGetHandler --- Setup a custom Get Handler
- SetNewHeadHandler --- Setup a custom Head Handler
- HTTP Password Processing --- Basic HTML Password Support
  - CheckAuthentication --- Checks the HTTP request for password information
  - RequestAuthentication --- Rejects the current HTTP request and request for a password

### HTML Header Functions

- SendHTMLHeader --- Sends an html response header
- SendHTMLHeaderWCookie --- Sends an html response header with a cookie
- SendTextHeader --- Sends a text header
- SendGifHeader --- Sends a gif header

### Other HTML Responses

- RedirectResponse --- Sends a response that redirects the request to the new page
- NotFoundResponse --- Sends a response that indicates the page can't be found

### Form Posting Functions

- ExtractPostData --- Decodes post data
- ExtractPostFile --- Extracts a file from the post data stream
- EnableMultiPartForms --- Enables multipart form posts
- DisableMultiPartForms --- Frees up the buffer

### Useful HTML/HTTP Functions

- writesafestring --- Writes out a string while escaping all special characters
- httpstricmp --- This is a special case insensitive prefix match compare

### HTML Encoded File Functions

- SendFullResponse --- Sends a complete encoded file as a response
- SendFileFragment --- Sends an encoded file as a part of a response

### Required Header file

```
#include <httppass.h>    // Found in C:\Nburn\include
```

## HTTP Password Support Functions

- CheckAuthentication --- Check the HTTP request for password information
- RequestAuthentication --- Rejects the current HTTP request and request for password

## 8.1. StartHTTP

### Synopsis:

```
void StartHTTP( WORD port=80 );
```

### Description:

This function starts the HTTP Server. **You must have initialized the IP stack before starting the HTTP Server.**

### Parameter:

| Type | Name | Description  |
|------|------|--|
| WORD | port | The port on which to open the HTTP Server. This defaults to the standard HTTP port (i.e. port 80). |

### Returns:

Nothing --- This is a void function

### Example:

Simple Html --- Found by default in C:\Nburn\examples



## 8.2. StopHTTP

### Synopsis:

```
void StopHTTP( );
```

### Description:

Use this function to shutdown the HTTP Server.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 8.3. SetNewPostHandler

### Synopsis:

```
typedef int ( http_posthandler )( int sock, PSTR url, PSTR pData, PSTR rxb );
```

```
http_posthandler * SetNewPostHandler( http_posthandler * newhandler );
```

### Description:

When the HTTP Server receives a POST request from an HTML form that request needs to be processed by a custom function, this call sets up that function. **Note:** This custom function **must** be of the form:

```
int yourposthandler( int sock, PSTR url, PSTR pData, PSTR rxb );
```

### Parameters:

| Type          | Name  | Description  |
|---------------|-------|--|
| int           | sock  | The File descriptor for the socket that this function should send a response to.               |
| PSTR (char *) | url   | The URL that this POST was directed at. This is used to select what form this was posted from. |
| PSTR          | pData | The encoded data fields from the form. See the ExtractPostData function to decode this data.   |
| PSTR          | rxb   | A pointer to the entire HTTP request. This is not usually needed.                              |

### Returns:

The function pointer to the previously registered post handler

### Example:

FlashForm --- Found by default in C:\Nburn\examples

## 8.4. SetNewGetHandler

### Synopsis:

```
typedef int ( http_gethandler )( int sock, PSTR url, PSTR rxb );

http_gethandler * SetNewGetHandler( http_gethandler * newhandler );
```

### Description:

When the HTTP Server receives a GET request, that request needs to be processed. The default GET processing looks for files stored in the compressed file data section by CompHtml. This allows **both** static and dynamic HTML responses; this is usually sufficient. If your application **needs** to have complete (program) control of GET requests (e.g. to implement Password protection), you may **replace** the **default** GET processing by registering a **new** function. **Note:** This custom function **must** be of the form:

```
int yourgethandler( int sock, PSTR url, PSTR rxb );
```

### Parameters:

| Type          | Name | Description   |
|---------------|------|---|
| int           | sock | The File descriptor for the socket that this function should send a response to.              |
| PSTR (char *) | url  | The URL that this GET was directed at. This is used to select what form this was posted from. |
| PSTR          | rxb  | A pointer to the entire HTTP request.   |

The default GET handler is as follows:

```
int BaseDoGet(int sock, PSTR url, PSTR rxBuffer)
{
    if (*url==0)    // The default value
    {
        RedirectResponse(sock,default_page);
        return 1;
    }
    if (httpstricmp(url,"ECHO"))
    {
        while (*url) url++;
        *url=' ';
        SendTextHeader(sock);
        writestring(sock,rxBuffer);
        return 1;
    }
    if(!SendFullResponse(url,sock))
    {
        // we failed
        NotFoundResponse(sock,url);
        DBPRINT(DB_HTTP,"Did not find:");
        DBPRINT(DB_HTTP,url);
    }
}
```

```
DBPRINT(DB_HTTP, "\r\n");  
}  
return 0;  
}
```

**Returns:**

The function pointer to the previously registered get handler

**Example:**

Simple HTML --- Found by default in C:\Nburn\examples

## 8.5. SetNewHeadHandler

### Synopsis:

```
typedef int ( http_headhandler )( int sock, PSTR url, PSTR rxb );  
  
http_headhandler * SetNewHeadHandler( http_headhandler * newhandle );
```

### Description:

When the HTTP server receives a HEAD, the default response does nothing. If you want to handle HEAD requests in any special way, you can write a custom function to do so. You **must** use this call to register this function.

The custom function **must** be of the form:

```
int yourheadhandler( int sock, PSTR url, PSTR rxb );
```

### Parameters:

| Type          | Name | Description  |
|---------------|------|--|
| int           | sock | The File descriptor for the socket that this function should send a response to. |
| PSTR (char *) | url  | The URL that this GET was directed at.   |
| PSTR          | rxb  | A pointer to the entire HTTP request.  |

### Returns:

The function pointer to the previously registered head handler

## 8.6. CheckAuthentication

### Synopsis:

```
int CheckAuthentication( PSTR url, char ** pPassword, char ** pUser );
```

### Description:

This password support function checks the HTTP request for password information. This function examines the PASSWORD field stored in the HTTP request.

### Parameters:

| Type | Name        | Description   |
|------|-------------|---|
| PSTR | url         | A pointer to the url as passed to the user provided get function.         |
| char | **pPassword | A char pointer that will be set to point to the password in the request.  |
| char | **pUser     | A char pointer that will be set to point to the user name in the request. |

### Example:

```
#include <http.h>
#include <httppass.h>
static http_gethandler * oldhand;
// somewhere in initialization...
oldhand=SetNewGetHandler(MyDoGet);
int MyDoGet(int sock, PSTR url, PSTR rxBuffer)
{
    char * pPass;
    char * pUser;
    if (!CheckAuthentication(url,&pPass,&pUser))
    {
        RequestAuthentication(sock,"YourNameHere");
        return 1;
    }
    else
    {
        if (password is not ok)
        {
            RequestAuthentication(sock,"YourNameHere");
            return 1;
        }
    }
    // If we got here the password was acceptable
    return (* oldhand)(sock,url,rxBuffer);
}
```

## 8.7. RequestAuthentication

### Synopsis:

```
void RequestAuthentication( int sock, PCSTR name );
```

### Description:

This password support function rejects the current HTTP request and request for password. This function **will** send a 401 authentication requested message to the socket.

### Parameters:

| Type  | Name | Description  |
|-------|------|--|
| int   | sock | The socket file descriptor to get the request.       |
| PCSTR | name | The Name that should appear in the password request. |

### Returns:

Nothing ---This is a void function

## 8.8. SendHTMLHeader

### Synopsis:

```
void SendHTMLHeader( int sock );
```

### Description:

This function sends an HTML header response to the selected socket. This function is used if you are building your own HTML response (programmatically) from scratch.

### Parameter:

| Type | Name | Description                                       |
|------|------|---|
| int  | sock | The socket that you want to send the Redirect to. |

### Returns:

Nothing --- This is a void function



## 8.9. SendHTMLHeaderWCookie

### Synopsis:

```
void SendHTMLHeaderWCookie( int sock, char * cookie );
```

### Description:

This function sends an HTML response header and includes a cookie to be stored by the browser.

### Parameters:

| Type | Name    | Description   |
|------|---------|---|
| int  | sock    | The socket that you want to send the Redirect to.       |
| char | *cookie | A char pointer that will be set to point to the cookie. |

### Returns:

Nothing ---This is a void function

### Example Application:

HtmlCookie --- Found in C:\Nburn\examples

## 8.10. SendTextHeader

### Synopsis:

```
void SendTextHeader( int sock );
```

### Description:

This function is used if you want to build a plain text response from scratch. **It should be sent before any other part of the HTTP response.**

### Parameter:

| Type | Name | Description                                       |
|------|------|---|
| int  | sock | The socket that you want to send the Redirect to. |

### Returns:

Nothing --- This is a void function

### Example:

The default GET response ECHO uses this function

## 8.11. SendGifHeader

### Synopsis:

```
void SendGifHeader( int sock );
```

### Description:

This function sends a GIF response header. This function is used to set the header type for stored GIF files in the system. This function can be used to generate a GIF header for dynamically generated GIF files.

### Parameter:

| Type | Name | Description                                       |
|------|------|---|
| int  | sock | The socket that you want to send the Redirect to. |

### Returns:

Nothing ---This is a void function

## 8.12. RedirectResponse

### Synopsis:

```
void RedirectResponse( int sock, PCSTR new_page );
```

### Description:

When an HTTP GET or POST request should be redirected to a different page, use this function to do so.

### Parameters:

| Type                 | Name     | Description                                       |
|----------------------|----------|---|
| int                  | sock     | The socket that you want to send the Redirect to. |
| PCSTR (const char *) | new_page | The URL of the redirected destination.            |

### Returns:

Nothing --- This is a void function

## 8.13. NotFoundResponse

### Synopsis:

```
void NotFoundResponse( int sock, PCSTR errored_page );
```

### Description:

This function responds to an HTTP GET or POST request by indicating the page does not exist.

### Parameters:

| Type                 | Name         | Description  |
|----------------------|--------------|--|
| int                  | sock         | The socket that you want to send the Redirect to.  |
| PCSTR (const char *) | errored_page | The URL of the requested page that does not exist. |

### Returns:

Nothing --- This is a void function

## 8.14. ExtractPostData

### Synopsis:

```
int ExtractPostData( PCSTR name, PCSTR data, PSTR dest_buffer, int
maxlen );
```

### Description:

This function takes the HTML post data sent to the DoPost function, extracts the data associated with a specific name, and returns it in dest\_buffer.

### Parameters:

| Type  | Name        | Description  |
|-------|-------------|--|
| PCSTR | name        | The name of the HTML form data element to extract the data from. |
| PCSTR | data        | The data pointer passed in to the process post function.         |
| PCSTR | dest_buffer | The location of the data after it has been extracted.            |
| int   | maxlen      | The maximum length of the string.                                |

### Returns:

-1 --- If no data of that name was found  
0 --- If the data field was presented, but empty  
Otherwise, the number of chars extracted and copied into dest\_buffer

### Example Application:

FlashForm --- Found by default in your C:\Nburn\examples directory

## 8.15. ExtractPostFile

### Synopsis:

```
ExtractPostFile( PCSTR name, PCSTR pData );
```

### Description:

This function extracts a file from the post data stream. **Note:** The stream **must** be closed just like any other stream.

### Parameters:

| Type  | Name  | Description  |
|-------|-------|--|
| PCSTR | name  | The name of the HTML form data element to extract the data from. |
| PCSTR | pData | The encoded data fields from the form.                           |

### Returns:

> 0 --- If successful

## 8.16. EnableMultiPartForms

### Synopsis:

```
BOOL EnableMultiPartForms( DWORD maxfile_size );
```

### Description:

This function enables multipart form posts and set the max file size you will accept. This function also mallocs the file buffer.

### Parameter:

| Type  | Name         | Description            |
|-------|--------------|------------------------|
| DWORD | maxfile_size | The maximum file size. |

### Returns:

True --- If successful



## 8.17. DisableMultiPartForms

### Synopsis:

```
void DisableMultiPartForms( );
```

### Description:

This function frees up the buffer.

### Parameters:

None

### Returns:

Nothing ---This is a void function

## 8.18. writesafestring

### Synopsis:

```
void writesafestring( int fd, PCSTR str );
```

### Description:

When sending HTML test responses certain characters (e.g. '<') are interpreted by the browser as formatting, not as text. **Note:** This function properly escapes the text so it **will** appear as desired, and sends it out the associated socket descriptor.

### Parameters:

| Type                 | Name | Description                                |
|----------------------|------|--|
| int                  | fd   | The file descriptor to send the string to. |
| PCSTR (const char *) | str  | The NULL terminated string to send.        |

### Returns:

The number of chars sent

## 8.19. httpstricmp

### Synopsis:

```
int httpstricmp( PCSTR s1, PCSTR sisupper2 );
```

### Description:

This function is used internally to match URLs with stored file prefixes.

### Parameters:

| Type  | Name      | Description  |
|-------|-----------|--|
| PCSTR | s1        | The string to test.  |
| PCSTR | sisupper2 | The reference string (which <b>must</b> already be all <b>uppercase</b> ). |

### Returns:

0 --- If the string prefixes do not match  
1 --- If the string prefixes do match

### Example:

`httpstricmp (s1, "LED.HTML" )` would return:

| s1 Value  | Return Value |
|-----------|--------------|
| LED       | 1            |
| led.HTML  | 1            |
| led.html? | 1            |
| LED.HTM?  | 0            |

## 8.20. SendFullResponse

### Synopsis:

```
int SendFullResponse( char * name, int fd );
```

### Description:

This function looks for the file named "name" in the files stored in the system with CompHtml. If it finds the file, it sends the proper HTTP header and renders the file to the socket. **Note:** If the stored file has embedded dynamic HTML, these functions will be filled in.

### Parameters:

| Type | Name  | Description  |
|------|-------|--|
| char | *name | The name of the file to send.                      |
| int  | fd    | The file descriptor or socket to send the file to. |

### Returns:

1 --- If the file was found and returned  
0 --- If the file was not found

## 8.21. SendFileFragment

### Synopsis:

```
int SendFileFragment( char * name, int fd );
```

### Description:

This function looks for the file named "name" in the files stored in the system with CompHtml. If it finds this file, it sends the file as a fragment. It does not send the HTTP header. If the stored file has embedded dynamic HTML, these functions will be filled in. If you want to build HTML responses with large chunks of pre-configured HTML, you can store these in the system and then send them out sequentially using this function.

### Parameters:

| Type | Name  | Description  |
|------|-------|--|
| char | *name | The name of the file fragment to send.             |
| int  | fd    | The file descriptor or socket to send the file to. |

### Returns:

1 --- If the file was found and returned  
0 --- If the file was not found

## 9. Interrupts

### 9.1. INTERRUPT MACRO

#### Required Header Files:

```
#include <ucos.h>          // Found in C:\Nburn\include
#include <cfinter.h>       // Found in C:\Nburn\include
```

**Warning:** cfinter.h must be included after ucos.h.

#### Synopsis:

```
INTERRUPT( Name, SRValue )
```

#### Description:

The INTERRUPT Macro sets up an Interrupt function and the code block that will do the necessary things to save and restore the CPU registers. In addition, this macro tells the RTOS that an Interrupt is happening.

**All Level 7 interrupts are non-maskable.**

#### Parameters:

| Type    | Description   |
|---------|---|
| Name    | The name of the Interrupt function.   |
| SRValue | The SR register value that you want the processor to have during the interrupt. |

The eight permitted SR Register Values are:

1. 0x2000 - **Allows all** interrupts
2. 0x2100 - **Blocks all** interrupts **below** Level 2
3. 0x2200 - **Blocks all** interrupts **below** Level 3
4. 0x2300 - **Blocks all** interrupts **below** Level 4
5. 0x2400 - **Blocks all** interrupts **below** Level 5
6. 0x2500 - **Blocks all** interrupts **below** Level 6
7. 0x2600 - **Blocks all** interrupts **below** Level 7
8. 0x2700 - **Blocks all** interrupts **below** Level 7

**Important:** The code **within** the INTERRUPT macro can be called at **any** time. Certain OS and I/O functions **cannot** be called from within an Interrupt routine.

**Warning:** The following functions are not legal within an interrupt routine

#### All UCOS Critical Section Functions

- USER\_ENTER\_CRITICAL
- USER\_EXIT\_CRITICAL
- UCOS\_ENTER\_CRITICAL
- UCOS\_EXIT\_CRITICAL

#### All UCOS Init or Pend Functions

- OSxxINIT
- OSxxPend
- OSCritEnter
- OSChangePrio
- OSTaskDelete
- OSLock
- OSUnlock
- OSTaskCreate
- OSTimeDly

**Note:** All OSxxPendNoWait functions should be OK.

#### I/O Functions from within the ISR (i.e. Interrupt Service Routine)

- write
- writeall
- read
- printf
- fprintf
- iprintf
- scanf
- gets
- puts

#### Memory Management Functions

- Malloc
- Free
- New
- Delete

**Important:** Once you used the INTERRUPT macro to define the interrupt function, you **will** need to use the Set Interrupt Controller Function (SetIntc) to set up your Interrupt Controller variables and point the interrupt vector at that function. Please refer to your Freescale MCF Hardware User's Manual (in **C:\Nburn\docs\platform**) for your **specific** vector number and controller number.

## 9.2. SetIntc (MCF5234 and MCF5282 Only)

### Synopsis:

```
void SetIntc( int intc, long func, int vector, int level, int prio )
```

### Usage:

```
extern "C"
{
    void SetIntc( int intc, long func, int vector, int level, int prio )
}
```

### Description:

This function sets up the Interrupt Controller variables.

### Parameters:

| Type | Name   | Description   |
|------|--------|---|
| int  | intc   | The interrupt controller number- either 0 (zero) or 1. This number can be found in your Freescale MCF Hardware User's Manual. |
| int  | func   | The function to call. <b>In the (hypothetical) usage example above, we used the name cast as a long.</b>                      |
| int  | vector | The vector number that can be found in your Freescale MCF Hardware User's Manual (in <b>C:\Nburn\docs\platform</b> ).         |
| int  | level  | The interrupt level you want to assign. <b>Note:</b> This value can be anything from 1 to 6.                                  |
| int  | prio   | The interrupt priority used to resolve who goes first if multiple interrupts at the same level are occurring.                 |

### Returns:

Nothing --- This is a void function



### 9.3. SetIntc (MCF5270 Only)

#### Synopsis:

```
void SetIntc( long func, int vector, int level, int prio )
```

#### Usage:

```
extern "C"
{
    void SetIntc( long func, int vector, int level, int prio )
}
```

#### Description:

This function sets up the Interrupt Controller variables.

#### Parameters:

| Type | Name   | Description   |
|------|--------|---|
| int  | func   | The function to call. <b>In the (hypothetical) usage example above, we used the name cast as a long.</b>              |
| int  | vector | The vector number that can be found in your Freescale MCF Hardware User's Manual (in <b>C:\Nburn\docs\platform</b> ). |
| int  | level  | The interrupt level you want to assign. <b>Note:</b> This value can be anything from 1 to 6.                          |
| int  | prio   | The interrupt priority used to resolve who goes first if multiple interrupts at the same level are occurring.         |

#### Returns:

Nothing --- This is a void function

## 9.4. Example for MCF5234 and MCF5282 Only

```

/* Include define the proper things */
#include <cfinter.h>
extern "C"
{
    void SetIntc( int intc, long func, int vector, int level, int prio )
}

/* Define the ISR function and SR Mask. The SR Mask can be 0x2?00 where
"? " is a value 0-7. The "? " value will block all interrupt at or below
this level. The "? " value must be at least as high as the level of the
interrupt defined in setintc & level 7 is always un-maskable */
INTERRUPT( My_ISR,0x2?00 )

{
    /* Put the meat of your interrupt routine here */
}

/* Initialize your interrupts (possibly start of main) */
SetIntc( ?, /* Interrupt controller 0 */
(long)My_ISR, /* The function to call */
?,/* Interrupt vector from Freescale's MCF User's Manual */
?, /* Interrupt level */
? );' /* Priority */

```

## 10. I/O System Library

### Required Header File

```
#include<iosys.h>          // Found in C:\Nburn\include
```

### General File Descriptor Functions

- close --- Close open file descriptors
- read --- Read data from a file descriptor
- ReadWithTimeout --- Read from a FD with timeout
- dataavail --- Check to see if data is available for read
- charavail --- Check to see if data is available for read on stdin
- write --- Write data to a file descriptor
- writestring --- Write a string to the file descriptor
- writeall --- Write data to a file descriptor and block until complete

### FD Set and Select Related Functions

- FD\_ZERO --- Zero a file descriptor set
- FD\_CLR --- Clear a specific fd in a fd\_set
- FD\_SET --- Set a specific fd in a fd\_set
- FD\_ISSET --- Test to see if a specific fd is set in an fd\_set
- select --- ZeroWaitSelect
- ZeroWaitSelect --- ZeroWaitSelect

### Standard I/O Modification Functions

- ioctl --- Control translation and formatting for stdio
- ReplaceStdio --- Replace (i.e. override) the stdio file descriptor with a new one

### Miscellaneous Function

### Required Header File

```
#include<syslog.h>        // Found in C:\Nburn\include
```

- SysLog --- Sends output on UDP Port 514

## 10.1. close

### Synopsis:

```
int close( int fd );
```

### Description:

This function closes the resources associated with a file descriptor (fd). This can be a TCP socket or a Serial I/O port.

### Parameter:

| Type | Name | Description                |
|------|------|----------------------------|
| int  | fd   | The file descriptor number |

### Returns:

0 (zero) --- On success

A resource specific error code --- On failure

### See Also:

- read --- Read data from a file descriptor
- ReadWithTimeout --- Read from a FD with timeout
- write --- Write data to a file descriptor
- writestring --- Write a string to the file descriptor

## 10.2. read

### Synopsis:

```
int read( int fd, char * buf, int nbytes );
```

### Description:

This function reads data from a file descriptor (fd), and will block forever until at least one byte is available to be read (as opposed to the ReadWithTimeout function which reads data from a file descriptor with a specified time-out value). This function can be used to read from stdio, TCP sockets, or Serial ports.

### Parameters:

| Type | Name   | Description                        |
|------|--------|------------------------------------|
| int  | fd     | The file descriptor number         |
| char | *buf   | A pointer to the read destination. |
| int  | nbytes | Maximum number of bytes to read.   |

### Returns:

The number of bytes read --- On success

A negative number (representing the resource specific error) --- On failure

### See Also:

- close --- Close open file descriptors
- ReadWithTimeout --- Read from a FD with timeout
- write --- Write data to a file descriptor
- writestring --- Write a string to the file descriptor

## 10.3. ReadWithTimeout

### Synopsis:

```
int ReadWithTimeout( int fd, char * buf, int nbytes, unsigned long
timeout );
```

### Description:

This function reads data from a file descriptor (fd), with a **specified time-out value** (as opposed to the **read function which will block forever until at least one byte is available to be read**). This function will block until **either** the time-out expires **or** at least one byte is available to be read. This function can be used to read from stdio, TCP sockets, or Serial ports.

**Important:** This function operates like a read function in that it reads **all** available bytes and returns. The addition of a time-out does **not** cause the function to block until the maximum number of bytes specified in the function call is available. As with read, the application **must** use the return value of the ReadWithTimeout function to determine how many bytes were read, and call the function again if necessary.

### Parameters:

| Type          | Name    | Description                                 |
|---------------|---------|---|
| int           | fd      | The file descriptor number                  |
| char          | *buf    | A pointer to the read destination.          |
| int           | nbytes  | Maximum number of bytes to read.            |
| unsigned long | timeout | The number of timer ticks to wait for data. |

### Returns:

0 (zero) --- On timeout

The number of bytes read --- On success

A negative number (representing the resource specific error) --- On failure

### See Also:

- close --- Close open file descriptors.
- read --- Read data from a file descriptor
- write --- Write data to a file descriptor
- writestring --- Write a string to the file descriptor

## 10.4. dataavail

### Synopsis:

```
int dataavail( int fd );
```

### Description:

This function checks to see if data is available for read.

### Parameter:

| Type | Name | Description                |
|------|------|----------------------------|
| int  | fd   | The file descriptor number |

### Returns:

1 --- If data is available  
0 --- If no data is available

### See Also:

- charavail --- Is data is available for read on stdin?
- read --- Read data from a file descriptor

## 10.5. charavail

### Synopsis:

```
int charavail( );
```

### Description:

This function checks to see if data is available for read on stdin.

### Parameters:

None

### Returns:

1 --- If data is available  
0 --- If no data is available

### See Also:

- dataavail --- Is data is available for read?
- read --- Read data from a file descriptor



## 10.6. write

### Synopsis:

```
int write( int fd, const char * buf, int nbytes );
```

### Description:

This function writes data to the stream associated with a file descriptor (fd). This function can be used to write data to stdio, a TCP socket, or a Serial port. Note: The write function **will** block until at least one byte is written, but does **not** have to write all the bytes requested. For example, if you wanted to write 100 bytes, and there was only room in the buffer for 5, then the write function would return 5.

### Parameters:

| Type       | Name   | Description                       |
|------------|--------|-----------------------------------|
| int        | fd     | The file descriptor number        |
| const char | *buf   | A pointer to the byte to write.   |
| int        | nbytes | Maximum number of bytes to write. |

### Returns:

The number of bytes written (**Note:** This value can be **less** than the number of bytes requested)  
0 (zero) --- If the write timed out  
A negative number --- If an error occurred

### See Also:

- close --- Close open file descriptors
- read --- Read data from a file descriptor
- ReadWithTimeout --- Read from a FD with timeout
- writestring --- Write a string to the file descriptor
- writesafestring --- Write out a string while escaping all special HTML characters
- writeall --- Write data to a file descriptor and block until complete

## 10.7. Writestring

### Synopsis:

```
int writestring( int fd, const char * str );
```

### Description:

This function writes null terminated string data to the stream associated with a file descriptor (fd). This function can be used to write data to stdio, a TCP socket, or a Serial port.

### Parameters:

| Type       | Name | Description                                       |
|------------|------|---|
| int        | fd   | The file descriptor number                        |
| const char | *str | A pointer to the NULL terminated string to write. |

### Returns:

The number of bytes written (Note: This value can be less than the number of bytes requested.)  
0 (zero) --- If the write timed out  
A negative number --- If an error occurred

### See Also:

- close --- Close open file descriptors
- read --- Read data from a file descriptor
- ReadWithTimeout --- Read from a FD with timeout
- write --- Write data to a file descriptor
- writesafestring --- Write out a string while escaping all special HTML characters

## 10.8. writeall

### Synopsis:

```
int writeall( int fd, const char * buf, int nbytes );
```

### Description:

This function writes data to the stream associated with a file descriptor (fd). This function can be used to write data to stdio, a TCP socket, or a Serial port. It will block and wait for the fd to either send the whole requested amount or to return an error.

### Parameters:

| Type       | Name   | Description                       |
|------------|--------|-----------------------------------|
| int        | fd     | The file descriptor number        |
| const char | *buf   | A pointer to the byte to write.   |
| int        | nbytes | Maximum number of bytes to write. |

### Returns:

The number of bytes written

A negative number --- If an error occurred

### See Also:

- close --- Close open file descriptors
- read --- Read data from a file descriptor
- ReadWithTimeout --- Read from a FD with timeout
- write --- Write data to a file descriptor
- writestring --- Write a string to the file descriptor
- writesafestring --- Write out a string while escaping all special HTML characters

## 10.9. FD\_ZERO

### Synopsis:

```
void FD_ZERO( fd_set * pfds );
```

### Description:

This function zero's a fd\_set (file descriptor set) so that it has no file descriptors (fds) selected.

### Parameter:

| Type   | Name  | Description             |
|--------|-------|-------------------------|
| fd_set | *pfds | A pointer to the fd_set |

### Returns:

Nothing --- This is a void function

### See Also:

- FD\_CLR --- Clear a specific fd in a fd\_set
- FD\_SET --- Set a specific fd in a fd\_set
- FD\_ISSET --- Test to see if a specific fd is set in an fd\_set
- select --- Wait for I/O events

## 10.10. FD\_CLR

### Synopsis:

```
void FD_CLR( int fd, fd_set * pfdset );
```

### Description:

A `fd_set` (file descriptor set) holds a set of file descriptors (fds). This function clears or removes a specific file descriptor in an `fd_set`.

### Parameters:

| Type   | Name    | Description                                     |
|--------|---------|---|
| int    | fd      | The file descriptor number.                     |
| fd_set | *pfdset | A pointer to the <code>fd_set</code> to modify. |

### Returns:

Nothing --- This is a void function

### See Also:

- FD\_ZERO --- Zero a file descriptor set
- FD\_SET --- Set a specific fd in a `fd_set`
- FD\_ISSET --- Test to see if a specific fd is set in a `fd_set`
- select --- Wait for I/O events

## 10.11. FD\_SET

### Synopsis:

```
void FD_SET( int fd, fd_set * pfdset );
```

### Description:

A fd\_set (file descriptor set) holds a set of file descriptors (fds). This function sets or adds a specific file descriptor to an fd\_set.

### Parameters:

| Type   | Name    | Description                        |
|--------|---------|------------------------------------|
| int    | fd      | The file descriptor number.        |
| fd_set | *pfdset | A pointer to the fd_set to modify. |

### Returns:

Nothing --- This is a void function

### See Also:

- FD\_ZERO --- Zero a file descriptor set
- FD\_CLR --- Clear a specific fd in a fd\_set
- FD\_ISSET --- Test to see if a specific fd is set in an fd\_set
- select --- Wait for I/O events

## 10.12. FD\_ISSET

### Synopsis:

```
int FD_ISSET( int fd, fd_set * pfdset );
```

### Description:

A fd\_set (file descriptor set) holds a set of file descriptors (fds). This function indicates whether (or not) a specific fd is in a specific fd\_set.

### Parameters:

| Type   | Name    | Description                      |
|--------|---------|----------------------------------|
| int    | fd      | The file descriptor number.      |
| fd_set | *pfdset | A pointer to the fd_set to test. |

### Returns:

0 (zero) --- If the fd is not in the set  
A non zero --- If the fd is in the set

### See Also:

- FD\_ZERO --- Zero a file descriptor set
- FD\_CLR --- Clear a specific fd in a fd\_set
- FD\_SET --- Set a specific fd in a fd\_set
- select --- Wait for I/O events

## 10.13. select

### Synopsis:

```
int select( int nfds, fd_set * readfds, fd_set * writefds, fd_set
* errorfds, unsigned long timeout );
```

### Description:

This function waits for events to occur on one or more I/O resources associated with a set of file descriptors (fds). The user indicates his/her interest in specific fds by setting them in the fd\_sets (file descriptor set) that are passed into the function. Note: This function will "unblock" when at least one byte is available for the file descriptor you add to the output set.

### Parameters:

| Type          | Name      | Description  |
|---------------|-----------|--|
| int           | nfds      | The number of file descriptors to examine. <b>Note:</b> This parameter is currently ignored  |
| fd_set        | *readfds  | A pointer to the fd_set to select for read events. <b>Note:</b> This parameter can be NULL. It is modified on exit to reflect the read availability of the selected fds in the set.                |
| fd_set        | *writefds | A pointer to the fd_set to select for write availability events. <b>Note:</b> This parameter can be NULL. It is modified on exit to reflect the write availability of the selected fds in the set. |
| fd_set        | *errorfds | A pointer to the fd_set to select for error events. <b>Note:</b> This parameter can be NULL. It is modified on exit to reflect the error state of the selected fds in the set.                     |
| unsigned long | timeout   | The number of time ticks to wait before timing out if no events occurred in the selected fd set.   |

### Returns:

The number of fds in all of the non null fd\_sets or 0 (zero) if the function timed out

### See Also:

- FD\_ZERO --- Zero a file descriptor set
- FD\_CLR --- Clear a specific fd in a fd\_set
- FD\_SET --- Set a specific fd in a fd\_set
- FD\_ISSET --- Test to see if a specific fd is set in an fd\_set



## 10.14. ZeroWaitSelect

### Synopsis:

```
int ZeroWaitSelect( int nfd, fd_set * readfds, fd_set * writefds,  
fd_set * errorfds )
```

### Description:

This function waits for events to occur on one or more I/O resources associated with a set of file descriptors (fds). The user indicates his/her interest in specific fds by setting them in the fd\_sets (file descriptor set) that are passed into the function.

### Parameters:

| Type   | Name      | Description  |
|--------|-----------|--|
| int    | nfd       | The number of file descriptors to examine. <b>Note:</b> This parameter is currently ignored  |
| fd_set | *readfds  | A pointer to the fd_set to select for read events. <b>Note:</b> This parameter can be NULL. It is modified on exit to reflect the read availability of the selected fds in the set.                |
| fd_set | *writefds | A pointer to the fd_set to select for write availability events. <b>Note:</b> This parameter can be NULL. It is modified on exit to reflect the write availability of the selected fds in the set. |
| fd_set | *errorfds | A pointer to the fd_set to select for error events. <b>Note:</b> This parameter can be NULL. It is modified on exit to reflect the error state of the selected fds in the set.                     |

### Returns:

The number of fds in all of the non null fd\_sets or 0 (zero) if there are no valid fds

### See Also:

FD\_ZERO --- Zero a file descriptor set  
FD\_CLR --- Clear a specific fd in a fd\_set  
FD\_SET --- Set a specific fd in a fd\_set  
FD\_ISSET --- Test to see if a specific fd is set in an fd\_set

## 10.15.     ioctl

### Synopsis:

```
int ioctl( int fd, int cmd );
```

### Description:

This function controls the selection of three options for stdio: stdin = 0, stdout = 1 and stderr = 2. The four legal options are:

1. IOCTL\_TX\_CHANGE\_CRLF (1) /\* When set transmitted char \n gets converted to \r\n \*/
2. IOCTL\_RX\_CHANGE\_CRLF (2) /\* When set received \r\n get turned into \n \*/
3. IOCTL\_RX\_PROCESS\_EDITS (4) /\* When set Process backspace and do simple line editing \*/
4. IOCTL\_RX\_ECHO (8) /\* When set echo chars received to tx\*/

### Parameters:

| Type | Name | Description  |
|------|------|--|
| int  | fd   | The file descriptor number. The three options are: <ul style="list-style-type: none"><li>• 0 = stdin</li><li>• 1 = stdout</li><li>• 2 = stderr</li></ul> |
| int  | cmd  | The ioctl command consists of IOCTL_SET or IOCTL_CLR and the bit of the associated options.  |

### Returns:

The old option value

### See Also:

ReplaceStdio --- Replace the stdio file descriptor with a new one.

## 10.16. ReplaceStdio

### Synopsis:

```
int ReplaceStdio( int stdio_fd, int new_fd );
```

### Description:

This function allows you to map stdio to any file descriptor (fd). If the file descriptor generates an error (like a closed TCP connection), then stdio will be remapped to a negative fd (this will cause stdio to generate errors). When this function is used to remap an errored stdio channel, then the error will be cleared.

### Parameters:

| Type | Name     | Optional | Description  |
|------|----------|----------|--|
| int  | stdio_fd | No       | The stdio file descriptor to map to (0 = stdin, 1 = stdout, and 2 = stderr).   |
| int  | new_fd   | No       | The file descriptor to replace stdio with.<br><b>Note:</b> A value of 0 returns stdio to the default debug monitor based traps |

### Returns:

The value of the fd for the previous stdio override  
0 (zero) --- If stdio had not been mapped previously

### See Also:

ioctl --- Control translation and formatting for stdio

## 10.17. SysLog

### Required Header File:

```
#include<syslog.h>      // Found in C:\Nburn\include
```

### Synopsis:

```
int SysLog( const char * format, ... );
```

### Description:

This function works very similar to a standard printf function, in regards to input arguments, return values, and resulting output. The output is sent via UDP port 514, and it can be sent as either a broadcast or a unicast message. This function is useful for users who have no serial ports, are out of serial ports, or desire to send the debug output to a remote location.

If you do **not** specify an IP address to send to, then the message **will** be broadcasted. **Note:** If you are in crowded network, or if you desire to send a message to a particular IP address, then you **will** need to add the following line to your **start-up code**:

```
SysLogAddress = AsciiToIp( "<Destination IP Address>" );
```

For example:

```
SysLogAddress = AsciiToIp( "10.1.1.228" );
```

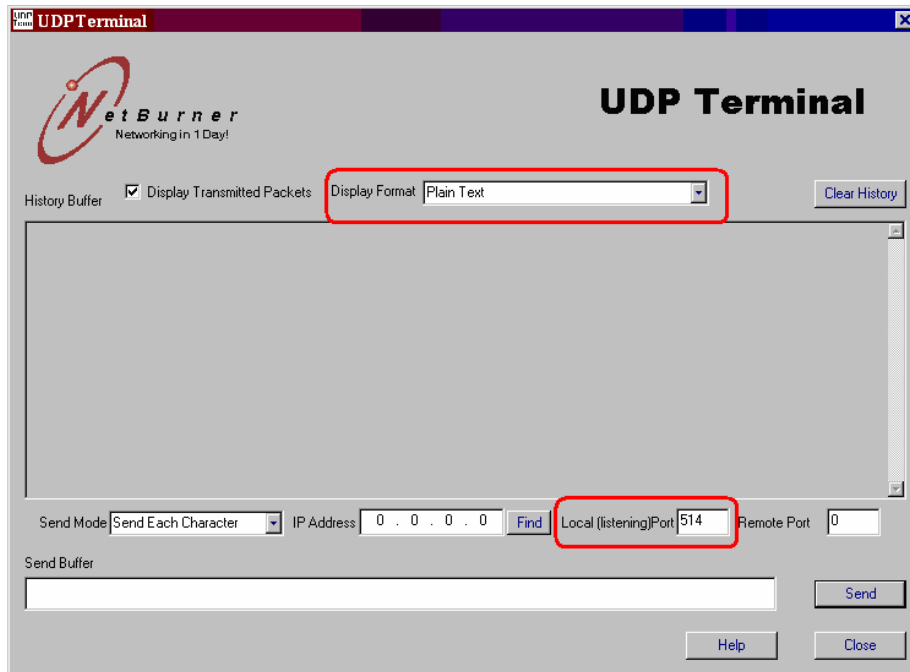
To make this function work in a normal project application, all you have to do is include the header file, and then in the program body, you use the SysLog() function as you would with an fprintf function. Some examples are:

```
SysLog( "Hello World!" );  
SysLog( "This number of seconds have passed: %d\r\n", Secs );  
SysLog( "%d plus %d equals %d.\r\n", num1, num2, sum );
```

To see the output generated by SysLog, open the **UDP Terminal Tool** application. From Windows: Start → Programs → Netburner NNDK → UDP Terminal Tool. By default this program is located in C:\Nburn\pcbin. Next, set the **Display Format** drop-down box to **Plain Text** and set the **Local (listening) Port** to **514** as shown on the next page.

If you have your NetBurner hardware connected to the network, and are periodically sending SysLog output, you **will** see the result in the UDP Terminal window. By default, it will broadcast **whatever** message is written to the network.

**Warning:** If a person is on a crowded network, sending broadcast messages to the SYSLOG port (i.e. Port 514) then using this function is a bad idea.



### Parameters:

The message (output string) to send followed by any number of variables - it works similar to the printf function.

### Returns:

The number of bytes in the output string, excluding the terminating NULL.

### Example:

```
...
#include <syslog.h>
...
...
void UserMain( void *pd )
{
    { StartUpCode }
    ...
    SysLogAddress = AsciiToIp( "10.1.1.228" ); // Only for unicasting
    while ( 1 )
    {
        OSTimeDly( TICKS_PER_SECOND );
        SysLog( "This is only a test! %d\r\n", Secs );
    }
}
```

## 11. I<sup>2</sup>C Library

### 11.1. Introduction

In the early 1980's, Philips Semiconductors developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter-IC or I<sup>2</sup>C-bus. Its name literally explains its purpose: to provide a communication link between Integrated Circuits. Its original purpose was to provide an easy way to connect a CPU to peripheral chips in a TV-set. All I<sup>2</sup>C-bus compatible devices incorporate an on-chip interface, which allows them to communicate directly with each other via the I<sup>2</sup>C-bus. This design concept solves the many interfacing problems encountered when designing digital control circuits. I<sup>2</sup>C has become a de facto world standard. The I<sup>2</sup>C-bus is patented by Philips.

The I<sup>2</sup>C-bus physically consists of two active wires and a ground connection. The active wires are called SDA and SCL. SDA is the Serial DATA line and SCL is the Serial CLOCK line. Every device hooked up to the bus has its own unique address. Each of these chips can act as a receiver and/or transmitter, depending on the functionality.

Both SDA and SCL are bi-directional signals, implemented in the following way : Each device on the I<sup>2</sup>C-bus can monitor the voltage or logic level on both signals. In addition, a device may connect the line to the system ground rail through an electronic switch or it may leave it floating. (For those of you who have done a bit of digital electronics, this is known as an 'open collector' output.) External resistors (typically about 4.7k ) are connected between each of the 2 signal lines and the +5V/3.3V power supply rail, so that if no device on the bus is connecting a line to ground, then that line appears to be in a logic 1 state.

The I<sup>2</sup>C-bus is a multi-master bus. This means that more than one IC capable of initiating a data transfer can be connected to it. The I<sup>2</sup>C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master (generally a microcontroller) and all the other ICs (at that time) are Bus Slaves. The most important thing to realize about data transfer on the I<sup>2</sup>C-bus is that the state of the SDA line can only change when the SCL line is in the logic low state. The reason for this is that the I<sup>2</sup>C protocol defines two special conditions to start and stop communications over the bus, they are:

1. A Start condition - defined as a change of SDA from logic 1 to logic 0 while SCL is high.
2. A Stop condition - defined as a change of SDA from logic 0 to logic 1 while SCL is high.

Prior to any transaction on the bus, a START condition needs to be issued on the bus by the Master. This start condition acts as a signal to all connected IC's that something is about to be transmitted on that bus. As a result, all connected chips will listen to the bus. The master controls the Clock line and always generates the Clock pulses.

Once a transmission is complete, the Master device can retain control of the bus by issuing a repeated start or RESTART condition. This gives the Master device the ability to immediately communicate with another device on the bus or to change transmission directions (read or write) with the current device.

After a message has been completed, a STOP condition is sent by the master. This is the signal for all devices on the bus that the bus is now available (i.e. idle). If a chip was accessed, and received data during the last transaction, it will now process that information (if it was not already processed during the reception of the message).

Devices on the I<sup>2</sup>C-bus are selected by an 8-bit address that is sent over the bus in the same way as data bytes. The least significant bit of this address acts as a read/write control signal and is set to 0 to make the Slave a Receiver and 1 to make the Slave a Transmitter. The address byte is the first byte transmitted after a Start condition. It is always transmitted by the master. By convention, if the slave is a receiver (and contains several registers), then the next byte transmitted after the address byte is an internal register address for that device. However, this is not required by the I<sup>2</sup>C specification.

## 11.2. The NetBurner I<sup>2</sup>C API

**Warning:** The only NetBurner hardware platforms that support I<sup>2</sup>C are:

- Mod5270 (I<sup>2</sup>C master and slave – through Hardware Peripherals)
- Mod5282 (I<sup>2</sup>C master and slave – through Hardware Peripherals)
- Mod5234 (I<sup>2</sup>C master and slave – through Hardware Peripherals)
- PK70 (I<sup>2</sup>C master and slave – through Hardware Peripherals)
- Mod5272 (I<sup>2</sup>C master only - Software emulated through Timers)

The NetBurner I<sup>2</sup>C API creates an easy interface to use interrupt driven I<sup>2</sup>C communication. The NetBurner I<sup>2</sup>C API comes in two flavors, I<sup>2</sup>C master and I<sup>2</sup>C multi.

The I<sup>2</sup>C master API allows you to configure the NetBurner device to act as the sole master device on an I<sup>2</sup>C bus while the I<sup>2</sup>C multi API configures the NetBurner device to be a multi-master device.

The I<sup>2</sup>C multi API should be used any time you wish to be able to have your NetBurner device act as both a bus master and slave device. This is very useful if you wish to have multiple smart-devices (e.g. NetBurner devices, Microcontrollers, etc.) on the same I<sup>2</sup>C bus. This API includes all the functions included in the I<sup>2</sup>C master API plus extra functions to handle slave mode transmissions and arbitration when attempting to become bus master.

The I<sup>2</sup>C master API is a limited functionality version of the I<sup>2</sup>C multi interface. Note: This mode will not work properly if there are other masters or multi-master devices on the same I<sup>2</sup>C bus. The master-only mode has the benefit of using less FLASH and RAM for space sensitive projects that only need to interface with slave devices. The space saved in FLASH is around 20 KB, and the space saved in RAM is dependent on the sizes configured for the slave mode RX/TX buffers.

### Required Header Files:

```
#include <i2cmulti.h>      // Found in C:\Nburn\<HWPlatform>\include
#include <i2cmaster.h>     // Found in C:\Nburn\<HWPlatform>\include
```

### Simple NetBurner I<sup>2</sup>C Functions

These functions are used in both I<sup>2</sup>C multi and master modes. They are simple in the fact that they require little configuration or code to send and receive buffers over I<sup>2</sup>C. Most devices that follow Philips I<sup>2</sup>C standard will work with these functions.

- I2CInit
- I2CSendBuf
- I2CReadBuf
- I2CRestart



## Simple NetBurner Slave Mode I<sup>2</sup>C Functions

These functions can only be implemented when the I<sup>2</sup>C multi driver is being used. These functions provide both slave RX and TX ability.

- I2CRXAvail
- I2CTXAvail
- I2CGetByte
- I2CFillSlaveTXBuf

## Advanced NetBurner I<sup>2</sup>C Functions

These functions are used in both I<sup>2</sup>C multi mode and master mode. They allow the user to control all aspects of I<sup>2</sup>C communication including start, stop, restart, and sending and receiving at single BYTE level. **Note:** These functions are useful when communicating with devices that do **not** follow Philips I<sup>2</sup>C standard exactly.

- I2CStart
- I2CStop
- I2CSend
- I2CRead

**Note:** An example I<sup>2</sup>C application can be found in C:\Nburn\examples\<HWPlatform>.

## 11.3. Simple I<sup>2</sup>C Functions

### 11.3.1. I2CInit

#### Synopsis:

```
void I2CInit( BYTE slave_Addr = 0x08, BYTE freqdiv = 0x11 );
```

**Note:** This version is used when including **i2cmulti.h**

#### Synopsis:

```
void I2CInit( BYTE freqdiv = 0x15 );
```

**Note:** This version is used when including **i2cmaster.h**

#### Description:

The I<sup>2</sup>C master and slave initialization routine must be called before performing any I<sup>2</sup>C functions.

#### Parameters:

| Type | Name       | Description  |
|------|------------|--|
| BYTE | slave_Addr | The 7-bit slave address assigned to NetBurner and is only used in i2cmulti API. <b>Note:</b> Phillips I <sup>2</sup> C Standard states that the two group addresses 0000XXX and 1111XXX are reserved for advanced purposes. The addresses 11110XX are also reserved for the 10-bit addressing I <sup>2</sup> C protocol.   |
| BYTE | freqdiv    | This value is used in the Coldfire I2FDR as a prescaler of the system clock to generate a max baud rate of the master mode I <sup>2</sup> C bus. Valid values for freqdiv are found in the I <sup>2</sup> C section of your Freescale MCF Manual (in C:\Nburn\docs). All the processors are set to be approximately 100Kbits by default, which is the max I <sup>2</sup> C standard speed. |

#### Returns:

Nothing --- This is a void function

### 11.3.2. I2CSendBuf

#### Synopsis:

```
BYTE I2CSendBuf( BYTE addr, PBYTE buf, int num, bool stop = true );
```

#### Description:

This function sends a buffer to an address on the I<sup>2</sup>C bus in master mode without the need of a start and stop bit.

#### Parameters:

| Type  | Name | Description  |
|-------|------|--|
| BYTE  | addr | The 7-bit address you wish to send the buffer to.  |
| PBYTE | buf  | A pointer to the BYTE buffer you wish to write from.   |
| int   | num  | The number of bytes to write.  |
| bool  | stop | True (default): Terminate communication with stop.<br>False: Do not terminate transmission. (This is useful if the user wishes to send a restart instead of a stop.) |

#### Returns:

The standard NetBurner I<sup>2</sup>C return values

### 11.3.3. I2CReadBuf

#### Synopsis:

```
BYTE I2CReadBuf( BYTE addr, PBYTE buf, int num, bool stop = true );
```

#### Description:

This function allows a buffer to be read from an address on the I<sup>2</sup>C bus in master mode without the need of a start and stop bit.

#### Parameters:

| Type  | Name | Description  |
|-------|------|--|
| BYTE  | addr | The 7-bit address you wish to read the buffer from.  |
| PBYTE | buf  | A pointer to the BYTE buffer you wish to read to.  |
| int   | num  | The number of bytes to read.   |
| bool  | stop | True (default): Terminate communication with stop.<br>False: Do not terminate transmission. (This is useful if the user wishes to send a restart instead of a stop.) |

#### Returns:

The standard NetBurner I<sup>2</sup>C return values

### 11.3.4. I2CRestart

#### Synopsis:

```
int I2CRestart( BYTE addr, BOOL Read_Not_Write, DWORD ticks_to_wait =  
I2C_RX_TX_TIMEOUT );
```

#### Description:

This function will send a restart signal on the I<sup>2</sup>C bus instead of a stop. **Note:** This function should **only** be used when in master mode **and** you have control of the bus.

#### Parameters:

| Type  | Name           | Description  |
|-------|----------------|--|
| BYTE  | addr           | The 7-bit address you wish to send the restart to.   |
| BOOL  | Read_Not_Write | True to read. False to write. <b>Note:</b> You can use I2C_START_READ or I2C_START_WRITE as values.                                |
| DWORD | ticks_to_wait  | The number of ticks to wait on restart before failing.<br><b>Note:</b> The default value in i2cmaster/multi.h = I2C_RX_TX_TIMEOUT. |

#### Returns:

The standard NetBurner I<sup>2</sup>C return values

## 11.4. Slave I<sup>2</sup>C Functions

### 11.4.1. I2CRXAvail

#### Synopsis:

```
bool I2CRXAvail( );
```

#### Description:

This function determines if there is data available in the I<sup>2</sup>C slave receive buffer.

#### Parameters:

None

#### Returns:

True --- If there is data in slave RX buffer  
False --- If slave RX buffer is empty

### 11.4.2. I2CTXAvail

**Synopsis:**

```
DWORD I2CTXAvail( );
```

**Description:**

This function determines the free space available in the I<sup>2</sup>C slave TX buffer.

**Parameters:**

None

**Returns:**

The number of bytes remaining in the I<sup>2</sup>C slave TX buffer

### 11.4.3. I2CGetByte

**Synopsis:**

```
BYTE I2CGetByte( );
```

**Description:**

This function will pend on a slave receive I<sup>2</sup>C semaphore.

**Parameters:**

None

**Returns:**

The oldest unread byte in the I<sup>2</sup>C slave RX buffer



#### 11.4.4. I2CFillSlaveTXBuf

##### Synopsis:

```
BYTE I2CFillSlaveTXBuf( PBYTE buf, DWORD num, bool restart = true );
```

##### Description:

This function is used to fill the I<sup>2</sup>C slave TX buffer.

##### Parameters:

| Type  | Name    | Description  |
|-------|---------|--|
| PBYTE | buf     | A pointer to the BYTE buffer that contains data for the slave TX buffer.   |
| DWORD | num     | The number of BYTE to send from buf to TX buffer.  |
| BOOL  | restart | If true then restart next TX from beginning TX buffer. (A new slave fill replaces buffer.) If false then continue next TX from last slave TX. (A new slave fill adds to buffer at last unread byte.) |

##### Returns:

Returns false if it failed to copy data

## 11.5. Advanced I<sup>2</sup>C Functions

### 11.5.1. I2CRead

#### Synopsis:

```
int I2CRead( PBYTE val, DWORD ticks_to_wait = I2C_RX_TX_TIMEOUT );
```

#### Description:

This function reads a single byte in master mode from the I<sup>2</sup>C bus.

#### Parameters:

| Type  | Name          | Description  |
|-------|---------------|--|
| PBYTE | val           | A pointer to byte location to store read value.          |
| DWORD | ticks_to_wait | The number of system time ticks before a timeout occurs. |

#### Returns:

The standard NetBurner I<sup>2</sup>C return values

### 11.5.2. I2CSend

#### Synopsis:

```
int I2CSend( BYTE val, DWORD ticks_to_wait = 5 );
```

#### Description:

This function sends a single byte in master mode on the I<sup>2</sup>C bus.

#### Parameters:

| Type  | Name          | Description  |
|-------|---------------|--|
| BYTE  | val           | The byte to send.  |
| DWORD | ticks_to_wait | The number of system time ticks before a timeout occurs. |

#### Returns:

The standard NetBurner I<sup>2</sup>C return values

### 11.5.3. I2CStart

#### Synopsis:

```
int I2CStart( BYTE addr, BOOL Read_Not_Write, DWORD ticks_to_wait =  
I2C_START_TIMEOUT );
```

#### Description:

This function is used to obtain an I<sup>2</sup>C bus and start communication in master mode.

#### Parameters:

| Type  | Name           | Description   |
|-------|----------------|---|
| BYTE  | addr           | The 7-bit address you wish to send the start to.  |
| BOOL  | Read_Not_Write | True to read. False to write. <b>Note:</b> You can use I2C_START_READ or I2C_START_WRITE as values                                |
| DWORD | ticks_to_wait  | The number of ticks to wait on restart before failing.<br><b>Note:</b> The default value in i2cmaster/multi.h = I2C_RX_TX_TIMEOUT |

#### Returns:

The standard NetBurner I<sup>2</sup>C return values

#### 11.5.4. I2CStop

**Synopsis:**

```
void I2CStop( );
```

**Description:**

This function ends communication and releases control of the I<sup>2</sup>C bus. This function puts your NetBurner board into idle/slave mode.

**Parameters:**

None

**Returns:**

Nothing --- This is a void function

## 12. Multicast Library

### 12.1. Introduction

This Module provides code for joining multicast groups. This module uses the IGMP protocol defined in RFC1112 and RFC 2236. IGMP Multicast is a method for distributing UDP packets within a group of hosts and servers.

The NetBurner Multicast functions extend the NetBurner UDP interface. Instead of using the RegisterUDPFifo function, you would use the RegisterMulticastFifo function to listen for Multicast UDP packets. **Note:** To transmit Multicast packets, just use the normal UDP Send with a multicast IP Address.

### Required Header File

```
#include <multicast.h>    // Found in C:\Nburn\include
```

### Multicast Group Functions

- RegisterMulticastFifo --- Register to join a Multicast group
- UnregisterMulticastFifo --- Register to leave a Multicast group

### Multicast Example

```
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <ucos.h>
#include <udp.h>
#include <multicast.h>
#include <autoupdate.h>
extern "C" {
void UserMain(void * pd);
}
// Make sure they're 4 byte aligned to keep the ColdFire happy
DWORD MultiTestStk[USER_TASK_STK_SIZE] __attribute__((aligned(4)));
IPADDR; // The multicast address
BOOL shutdown;
// The UCOS task function that just sits and waits for Multi packets
void MultiReaderMain(void * pd)
{
int port=(int )pd;
printf("Reading from port %d\n",port);
OS_FIFO fifo;
OSFifoInit(&fifo);
// Register to listen for Multi packets on port number 'port'
RegisterMulticastFifo(ipaddr,port,&fifo);
while (!shutdown)
{
```

```

// We construct a UDP packet object using the FIFO
// This constructor will only return when we have received a packet
UDPPacket upkt(&fifo,10);
// Did we get a valid packet? or just time out?
if (upkt.Validate())
{WORD len=upkt.GetDataSize();
printf("Got UDP packet with %d Bytes From :",(int)len);
ShowIP(upkt.GetSourceAddress());
printf("\n");
ShowData(upkt.GetDataBuffer(),len);
printf("\n");
}
}
iprntf("Unregistering from group\r\n");
UnregisterMulticastFifo(ipaddr,port);
iprntf("Done unregistering\r\n");
}
void UserMain(void * pd)
{
int portnum;
char buffer[80];
InitializeStack();
EnableAutoUpdate();
printf("Multicast Test \n");
printf("Input the port number?\n");
scanf("%d",&portnum);
printf("\nEnter the Multicast IP Address?");
buffer[0]=0;
while(buffer[0]==0)
{
gets(buffer);
}
ipaddr=AsciiToIp(buffer);
printf("Listening/Sending on Port %d of Group:",portnum);
ShowIP(ipaddr);
printf("\n");
OSChangePrio(MAIN_PRIO);
OSTaskCreate(MultiReaderMain,(void
*)portnum,&MultiTestStk[USER_TASK_STK_SIZE] ,MultiTestStk,MAIN_PRIO-
1);
while(1)
{
iprntf("Enter the test to send in the packet. (Empty string to
unregister listener)\n");
gets(buffer);
if (strlen(buffer)==0)
{
iprntf("Un registering the listener\n");
iprntf("You must reset the board to continue\n");
shutdown=TRUE;
while(1) OSTimeDly(20);
}
else
{
printf("Sending %s on UDP port %d to IP Address ",buffer,portnum);
ShowIP(ipaddr);
{

```

```
UDPPacket pkt;  
pkt.SetSourcePort(portnum);  
pkt.SetDestinationPort(portnum);  
pkt.AddData(buffer);  
pkt.AddDataByte(0);  
pkt.Send(ipaddr);  
}  
printf("\n");  
}  
};  
}
```



## 12.2. RegisterMulticastFifo

### Synopsis:

```
void RegisterMulticastFifo( IPADDR group, WORD dest_port, OS_FIFO
*pfifo );
```

### Description:

This call initializes the Multicast system. Register to join a Multicast group. **Note:** It is **not** active until at least **one** join has taken place.

### Parameters:

| Type    | Name      | Description                            |
|---------|-----------|--|
| IPADDR  | group     | The IP Address of the group to join.   |
| WORD    | dest_port | The UDP Port to listen on.             |
| OS_FIFO | *pfifo    | The fifo to put incoming packets into. |

### Returns:

Nothing --- This is a void function

## 12.3. UnregisterMulticastFifo

### Synopsis:

```
void UnregisterMulticastFifo( IPADDR group, WORD destination_port );
```

### Description:

This call removes the NetBurner device from the specified multicast group.

### Parameters:

| Type   | Name      | Description                          |
|--------|-----------|--------------------------------------|
| IPADDR | group     | The IP Address of the group to leave |
| WORD   | dest_port | The UDP Port to listen on.           |

### Returns:

Nothing --- This is a void function

## 13. NBTime Library

### 13.1. Introduction

The NBTime Library allows users to set a system time using either an NTP Server or the Real Time Clock. The system time can also be set/read manually.

**Warning:** If you include `nbtime.h` in your application code, you can not include `time.h` and vice versa.

#### Required Header File

```
#include <nbtime.h>          // Found in C:\Nburn\include
```

#### Functions

- `time` --- Reads the System time
- `set_time` --- Sets the System time
- `GetNTPTime` --- Gets time from an NTP Server
- `SetNTPTime` --- Gets time from an NTP Server and sets the System time

**Warning:** The two platform specific functions below can only be used on NetBurner V1.0x Carrier boards or on NetBurner hardware with the same RTC setup as the NetBurner Carrier board.

- `IOBoardRTCSetRTCfromSystemTime` --- Sets the RTC time using the current System Time **Replaced by `RTCSetRTCfromSystemTime` and requires `rtc.h`**. This header file is found in `C:\Nburn\<HardwarePlatform>\include`.
- `IOBoardRTCSetSystemFromRTCTime` --- Sets the System time using the Real Time Clock **Replaced by `RTCSetSystemFromRTCTime` and requires `rtc.h`**. This header file is found in `C:\Nburn\<HardwarePlatform>\include`

## 13.2. time

### Synopsis:

```
time_t time( time_t * pt )
```

### Description:

This function reads the system time, and if the \*pt is not null it will write the time to this location.

**Warning:** If you include nbtime.h in your application code, you can not include time.h and vice versa.

### Parameter:

| Type   | Name | Description  |
|--------|------|--|
| time_t | *pt  | A pointer to a time_t structure that will be written with the current system time. Can be NULL if the user wishes not to write the time to a new structure |

### Returns:

The current set system time in time\_t format

### 13.3. set\_time

#### Synopsis:

```
time_t set_time( time_t time_to_set )
```

#### Description:

This function sets the system time with the value of time\_to\_set.

**Warning:** If you include nbtime.h in your application code, you cannot include time.h and vice versa.

#### Parameter:

| Type   | Name        | Description   |
|--------|-------------|---|
| time_t | time_to_set | A time_t structure that will be written to the current system time. |

#### Returns:

The current set system time in time\_t format

## 13.4. GetNTPTime

### Synopsis:

```
DWORD GetNTPTime( IPADDR NTP_server_ip )
```

### Description:

This function gets time from an NTP Server.

**Warning:** If you include nbtime.h in your application code, you cannot include time.h and vice versa.

### Parameter:

| Type   | Name          | Description  |
|--------|---------------|--|
| IPADDR | NTP_server_ip | The IP Address of the NTP Server that you want to get the time from. |

### Returns:

The NTP time to the nearest second --- If successful  
0 --- If it fails

## 13.5. SetNTPTime

### Synopsis:

```
BOOL SetNTPTime( IPADDR ntpserver )
```

### Description:

This function gets time from an NTP Server and sets the system time.

**Warning:** If you include `nbtime.h` in your application code, you cannot include `time.h` and vice versa.

### Parameter:

| Type   | Name      | Description  |
|--------|-----------|--|
| IPADDR | ntpserver | The IP Address of the NTP Server that you want to get the time from. |

### Returns:

TRUE --- If successful  
FALSE --- If it fails

## 13.6. IOBoardRTCSetRTCfromSystemTime

### Synopsis:

```
int IOBoardRTCSetRTCfromSystemTime( );
```

### Description:

This platform specific function sets the RTC time using the current system time.

**This function has been replaced by RTCSetRTCfromSystemTime.**

**Warning:** If you include nbtime.h in your application code, you cannot include time.h and vice versa.

### Parameters:

None

### Returns:

0 --- If successful



## 13.7. IOBoardRTCSetSystemFromRTCTime

### Synopsis:

```
IOBoardRTCSetSystemFromRTCTime( );
```

### Description:

This platform specific function sets the system time using the Real Time Clock.

**This function has been replaced by RTCSetSystemFromRTCTime.**

**Warning:** If you include nbtime.h in your application code, you cannot include time.h and vice versa.

### Parameters:

None

### Returns:

0 --- If successful

## 14. POP3 and E-Mail Libraries

### 14.1. Introduction

This Module provides code for sending, reading, and managing mail from a POP3 server. POP3 is defined in the document RFC1939.

#### Required Header Files

```
#include<pop3.h>           // Found in C:\Nburn\include
#include <mailto.h>         // Found in C:\Nburn\include
```

#### POP3 Client Functions

- **Initialization Functions**
  - POP3\_InitializeSession --- Initialize the POP3 network connection
  - POP3\_CloseSession --- Close the POP3 network connection
- **POP3 Command Functions**
  - POP3\_StatCmd --- Get the status of the Mailstore on the POP3 server
  - POP3\_ListCmd --- Get the size of the specified message
  - POP3\_DeleteCmd --- Delete a specific message on the server
  - POP3\_RetrieveMessage --- Retrieve a specific message from the server
  - GetPOPErrorString --- Returns the Error text for a specific code

#### SendMail Functions

- SendMail --- Send an E-Mail using the selected POP server
- SendMailEx --- Send an E-Mail using the selected POP server
- SendMailAuth --- Send an E-Mail using the selected POP server and a password

## POP3 Example

```
#include <pop3.h>
#include <dns.h>
#define USERID "username"
#define USERPASS "password"
#define SERVERNAME "pop.yourserver.com"
#define POP_PORT (110)
int StartSession()
{
    IPADDR srvr_addr;
    if (GetHostByName(SERVERNAME,
        &srvr_addr,0,TICKS_PER_SECOND*10)==DNS_OK)
    {
        printf("Got Server IP = "); ShowIP(srvr_addr); printf("\r\n");
        // Create the POP 3 session with the server
        int
        session=POP3_InitializeSession(srvr_addr,POP_PORT,USERID,USERPASS,TICK
        S_PER_SECOND*10);
        return session;
    }
    else
    {
        printf("Failed to get Server IP Address\n");
        return 0;
    }
}
#define MSG_BUF_SIZ (16000)
static char messagebuffer[MSG_BUF_SIZ];
void GetMessages()
{
    DWORD num_mess;
    DWORD num_bytes;
    int session=StartSession();
    if (session>0)
    {
        int rv=POP3_StatCmd(session, &num_mess, &num_bytes,
            10*TICKS_PER_SECOND);
        if (rv==POP_OK)
        {
            printf("The server has %ld messages and %ld bytes\r\n",num_mess,
                num_bytes);
            if (num_mess==0) return;
            for (DWORD nmsg=1; nmsg<=num_mess; nmsg++)
            {
                char * psub;
                char * pbody;
                DWORD predict_size;
                rv=POP3_ListCmd(session,nmsg, &predict_size,TICKS_PER_SECOND*10);
                printf("Predicted message size is %ld\r\n",predict_size);
                rv=POP3_RetrieveMessage(session,nmsg,messagebuffer,&psub,&pbody,MSG_B
                UF_SIZ-1,TICKS_PER_SECOND*20);
                if (rv>0)
                {
                    printf("Received a message of %d bytes\r\n",rv);
                }
            }
        }
    }
}
```

```
messagebuffer[rv]=0;
if (pbody)
{
printf("Body Size %ld\r\n,<Start of Message:>\r\n %s\r\n<End of
Message>\r\n",strlen(pbody),pbody);
}
else
{
printf("Unable to locate body.\r\nPrinting the entire message
\r\n<Start of Message:>\r\n %s\r\n<End of Message>\r\n",pbody);
}
char c;
do
{
printf("Delete this message (Y/N?)");
c=toupper(getchar());
}
while ((c!='N') && (c!='Y'));
if (c=='Y')
{
rv=POP3_DeleteCmd(session,nmsg,TICKS_PER_SECOND*10);
if (rv==POP_OK) printf("Message deleted\r\n");
else
printf("Delete command processing failed with
error:%s\r\n",GetPOPErrorString(rv));
}
else
printf("Message not deleted\r\n");
}
else
printf("Retrieve command processing failed with
error:%s\r\n",GetPOPErrorString(rv));
}
}
else
{
printf("STAT command processing failed with
error:%s\r\n",GetPOPErrorString(rv));
}
POP3_CloseSession(session);
}
else
{
printf("Failed to create session with
error:%s\r\n",GetPOPErrorString(session));
}
}
```

## 14.2. POP3\_InitializeSession

### Synopsis:

```
int POP3_InitializeSession( IPADDR server_address, WORD port, PCSTR
UserName, PCSTR PassWord, DWORD time_out );
```

### Description:

This function initializes the POP3 network connection. This call makes the connection to the POP Server and logs in with the Username and Password.

### Parameters:

| Type   | Name           | Description                           |
|--------|----------------|---------------------------------------|
| IPADDR | server_address | The IP Address of the Server.         |
| WORD   | port           | The port to connect to on the Server. |
| PCSTR  | UserName       | The account Username.                 |
| PCSTR  | PassWord       | The account Password.                 |
| DWORD  | time_out       | The number of ticks to wait.          |

### Return Values:

int --- The command success code  
> 0 --- Mail session  
POP\_TIMEOUT --- Time out  
POP\_PASSWORDERROR --- Network error  
POP\_CONNECTFAIL--- Password error  
POP\_NETWORKERROR --- Network error

## 14.3. POP3\_CloseSession

### Synopsis:

```
Int POP3_CloseSession( int session );
```

### Description:

This function closes the POP3 network connection. This function also flushes deleted messages. (See RFC1939 for additional information.)

### Parameter:

| Type | Name    | Description      |
|------|---------|------------------|
| int  | session | The POP3 session |

### Return Values:

int --- The command success code  
POP\_OK --- Closed successfully  
POP\_TIMEOUT --- Time out  
POP\_COMMANDFAIL --- Command error  
POP\_NETWORKERROR --- Network error  
POP\_BADSESSION--- Bad session number

## 14.4. POP3\_StatCmd

### Synopsis:

```
int POP3_StatCmd( int session, DWORD * num_messages, DWORD
* total_bytes, DWORD time_out );
```

### Description:

This function gets the status of the Mailstore on the POP3 server and retrieves the state of the mail store associated with this session.

### Parameters:

| Type  | Name          | Description   |
|-------|---------------|---|
| int   | session       | The POP3 session.   |
| DWORD | *num_messages | The DWORD variable to hold the number of pending messages.                    |
| DWORD | *total_bytes  | The DWORD variable to hold the total number of bytes in the pending messages. |
| DWORD | time_out      | The number of ticks to wait.  |

### Return Values:

int --- The command success code  
POP\_OK --- Command OK  
POP\_TIMEOUT --- Time out  
POP\_COMMANDFAIL --- Command error  
POP\_NETWORKERROR --- Network error  
POP\_BADSESSION--- Bad session number

## 14.5. POP3\_ListCmd

### Synopsis:

```
int POP3_ListCmd( int session, DWORD message_number, DWORD  
* total_bytes, DWORD time_out );
```

### Description:

This function gets the size of the specified message and retrieves the size of the message.

### Parameters:

| Type  | Name           | Description   |
|-------|----------------|---|
| int   | session        | The POP3 session.   |
| DWORD | message_number | Retrieves the size of the message.  |
| DWORD | *total_bytes   | The DWORD variable to hold the total number of bytes in the pending messages. |
| DWORD | time_out       | The number of ticks to wait.  |

### Return Values:

int --- The command success code  
POP\_OK --- Command OK  
POP\_TIMEOUT --- Time out  
POP\_COMMANDFAIL --- Command error  
POP\_NETWORKERROR --- Network error  
POP\_BADSESSION--- Bad session number



## 14.6. POP3\_DeleteCmd

### Synopsis:

```
int POP3_DeleteCmd( int session, DWORD message_number, DWORD time_out
);
```

### Description:

This function deletes a specific message on the server. **Note:** The message is **not** actually deleted until the session is closed.

### Parameters:

| Type  | Name           | Description                  |
|-------|----------------|------------------------------|
| int   | session        | The POP3 session.            |
| DWORD | message_number | The message to delete.       |
| DWORD | time_out       | The number of ticks to wait. |

### Return Values:

int --- The command success code  
POP\_OK --- Command OK  
POP\_TIMEOUT --- Time out  
POP\_COMMANDFAIL --- Command error  
POP\_NETWORKERROR --- Network error  
POP\_BADSESSION--- Bad session number

## 14.7. POP3\_RetrieveMessage

### Synopsis:

```
int POP3_RetrieveMessage( int session, DWORD message_number, char
* buffer, char ** subject_ptr, char ** body_ptr, int max_bufferlen,
DWORD time_out );
```

### Description:

This function retrieves a specific message from the server. The message is retrieved as a large block with all of the headers first. Note: The message is left on the server and will not be deleted until you call POP3\_DeleteCmd.

### Parameters:

| Type  | Name           | Description   |
|-------|----------------|---|
| int   | session        | The POP3 session.   |
| DWORD | message_number | The message to retrieve.  |
| char  | *buffer        | The buffer to hold the message.   |
| char  | **subject_ptr  | If not NULL, the char pointer will be left pointing at the message subject. |
| char  | **body_ptr     | If not NULL, the char pointer will be left pointing at the message body.    |
| int   | max_bufferlen  | The maximum size of the retrieved message.                                  |
| DWORD | time_out       | The number of ticks to wait.  |

### Return Values:

int --- The command success code  
 > 0 --- The length of the message retrieved  
 POP\_TIMEOUT --- Time out  
 POP\_COMMANDFAIL --- Command error  
 POP\_NETWORKERROR --- Network error  
 POP\_BADSESSION --- Bad session number

## 14.8. GetPOPErrorString

### Synopsis:

```
PCSTR GetPOPErrorString( int err );
```

### Description:

This function returns the error text for a specific code. **Warning:** This function only works for POP3 errors.

### Parameter:

| Type | Name | Description    |
|------|------|----------------|
| int  | err  | The error code |

### Return Value:

The text string

## 14.9. SendMail

### Synopsis:

```
int SendMail( IPADDR pop_server, PCSTR userid, PCSTR from_addr, PCSTR
to_addr, PCSTR subject, PCSTR textbody );
```

### Description:

This function sends an E-Mail using the selected POP server. The difference between this function and SendMailEx are the parameters used to create the E-Mail. The difference between this function and SendMailAuth is the password parameter.

**Note:** This function is **not** called unless you **create** an application that uses it to send mail.

### Parameters:

| Type   | Name       | Description  |
|--------|------------|--|
| IPADDR | pop_server | The IP Address of the POP Server to use.                 |
| PCSTR  | userid     | The ASCII string to provide for RFC931 Identification.   |
| PCSTR  | from_addr  | The "from" E-Mail address.                               |
| PCSTR  | to_addr    | The "to" E-Mail address (i.e. where to send the E-Mail). |
| PCSTR  | subject    | The E-Mail subject.                                      |
| PCSTR  | textbody   | The body of the E-Mail.                                  |

### Return Values:

0 --- If it fails  
1 --- If successful

## 14.10. SendMailEx

### Synopsis:

```
int SendMailEx( IPADDR pop_server, PCSTR userid, PCSTR
from_addr_rev_path, PCSTR from_addr_memo_hdr, PCSTR to_addr, PCSTR
subject, PCSTR textbody );
```

### Description:

This function sends an E-Mail using the selected POP server. The Ex stands for Extended. The difference between this function and SendMail are the parameters used to create the E-Mail.

**Note:** This function is **not** called unless you **create** an application that uses it to send mail.

### Parameters:

| Type   | Name               | Description  |
|--------|--------------------|--|
| IPADDR | pop_server         | The IP Address of the POP Server to use.                 |
| PCSTR  | userid             | The ASCII string to provide for RFC931 Identification.   |
| PCSTR  | from_addr_rev_path | The "from" E-Mail address - RFC 821 - <reverse-path>.    |
| PCSTR  | from_addr_memo_hdr | The "from" E-Mail address - RFC 821- memo header.        |
| PCSTR  | to_addr            | The "to" E-Mail address (i.e. where to send the E-Mail). |
| PCSTR  | subject            | The E-Mail subject.                                      |
| PCSTR  | textbody           | The body of the E-Mail.                                  |

### Return Values:

0 --- If it fails  
1 --- If successful

## 14.11. SendMailAuth

### Synopsis:

```
int SendMailAuth( IPADDR pop_server, PCSTR userid, PCSTR pass, PCSTR
from_addr, PCSTR to_addr, PCSTR subject, PCSTR textbody );
```

### Description:

This function sends an E-Mail using the selected POP server and a password. The difference between this function and the SendMail function is the password (pass) parameter.

### Parameters:

| Type   | Name       | Description  |
|--------|------------|--|
| IPADDR | pop_server | The IP Address of the POP Server to use.                 |
| PCSTR  | userid     | The ASCII string to provide for RFC931 Identification.   |
| PCSTR  | pass       | The ASCII String to provide for AUTH Identification.     |
| PCSTR  | from_addr  | The "from" E-Mail  |
| PCSTR  | to_addr    | The "to" E-Mail address (i.e. where to send the E-Mail). |
| PCSTR  | subject    | The E-Mail subject.                                      |
| PCSTR  | textbody   | The body of the E-Mail.                                  |

### Return Values:

0 --- If it fails  
1 --- If successful

## 15. RTC Library

### 15.1. Introduction

The NetBurner RTC Library acts as an interface between the Real Time Clock on your NetBurner device (**Mod5234, Mod5272, Mod5282, Mod5270, and NBPK70 only**) and the C Library System Time Function. For more information on time.h, please refer to (GNU) libc.pdf in C:\Nburn\docs.

**Note:** An example (named time) can be found in C:\Nburn\examples\<HardwarePlatform>.

### Required Header File

```
#include <rtc.h>      // Found in C:\Nburn\<HardwarePlatform>\include

struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

### Parameters

| Type | Parameter | Description   |
|------|-----------|---|
| int  | tm_yday   | The day of the year, from 0 to 366.                   |
| int  | tm_mon    | The month, from 1 to 12.                              |
| int  | tm_mday   | The day of the month, from 1 to 31.                   |
| int  | tm_year   | The year, from 2000 to 2099.                          |
| int  | tm_hour   | The hour, from 0 to 23.                               |
| int  | tm_min    | The minute, from 0 to 59.                             |
| int  | tm_sec    | The second, from 0 to 59.                             |
| int  | tm_wday   | The day of the week, from 0 (Sunday) to 6 (Saturday). |
| int  | tm_isdst  | Is Daylight Savings Time in effect?                   |

### Functions

- RTCGetTime --- Gets the current clock time
- RTCSetTime --- Sets the current clock time
- RTCSetSystemFromRTCTime --- Sets the system time using the RTC
- RTCSetRTCfromSystemTime --- Sets the RTC time using the current system time
- SetNTPTime --- Sets the current clock time (time received from an NTP server)

## 15.2. RTCGetTime

### Synopsis:

```
int RTCGetTime( BasicTimeStruct & bts );
```

### Description:

This function gets the current clock time.

### Parameter:

| Type            | Name | Description  |
|-----------------|------|--|
| BasicTimeStruct | &bts | The basic time struct to fill in with values from the clock. |

### Returns:

0 (zero) --- On success  
-1 --- On failure



## 15.3. RTCSetTime

### Synopsis:

```
int RTCSetTime( struct tm & bts );
```

### Description:

This function sets the current time in the clock.

### Parameter:

| Type      | Name | Description                                    |
|-----------|------|--|
| Struct tm | &bts | The basic time struct to use to set the clock. |

### Returns:

0 (zero) --- On success  
-1 --- On failure

## 15.4. RTCSetSystemFromRTCTime

### Synopsis:

```
int RTCSetSystemFromRTCTime( );
```

### Description:

This function sets the system time using the Real Time Clock.

### Parameters:

None

### Returns:

0 (zero) --- On success

## 15.5. RTCSetRTCfromSystemTime

### Synopsis:

```
int RTCSetRTCfromSystemTime( );
```

### Description:

This function sets the Real Time Clock time using the current system time.

### Parameters:

None

### Returns:

0 (zero) --- On success

## 15.6. SetNTPTime

### Required Header File:

```
include <nptime.h>           // Found in C:\Nburn\include
```

### Synopsis:

```
BOOL SetNTPTime( IPADDR ntpserver );
```

### Description:

This function sets the time from a NTP Server.

### Parameter:

| Type   | Name      | Description                                   |
|--------|-----------|---|
| IPADDR | ntpserver | The (numerical) IP Address of the NTP Server. |

### Returns:

True --- On Success

False --- On Failure

## 16. Serial Library

### 16.1. Introduction

All of the I/O functions in the NetBurner I/O System Library work with Serial ports. When your NetBurner device boots up, the serial ports are running in a polled mode. Calling the `OpenSerial` or the `SimpleOpenSerial` function will open the serial port in an interrupt driven and buffered mode. This will result in an increase in performance. If you want to enable the interrupt driven mode for the default debug port (UART 0), you would call `SerialClose` followed by `OpenSerial` or `SimpleOpenSerial`. Once `SerialClose` is called, UART 0 will no longer be connected to `stdio` (`stdin`, `stdout`, `stderr`). If you are using `stdio` calls such as `iprintf`, `printf`, `siprintf`, `sprintf`, etc., that rely on `stdio`, you must reconnect `stdio` to the appropriate serial port with these function calls, assuming `fdSerial` is the return value from `OpenSerial`) or `SimpleOpenSerial`.

```
SerialClose( 0 );           // close UART 0
fdSerial = OpenSerial( 0, 115200, 1, 8, eParityNone );
ReplaceStdio( 0, fdSerial ); // stdin
ReplaceStdio( 1, fdSerial ); // stdout
ReplaceStdio( 2, fdSerial ); // stderr
```

#### Required Header File:

```
#include<serial.h>          // Found in C:\Nburn\include
```

#### For ALL NetBurner Platforms

- `OpenSerial` --- Opens one of the Serial ports
- `SimpleOpenSerial` --- Macro to open one of the Serial ports
- `SerialClose` --- Closes a Serial port. (Note: The close function works fine as well)
- `SerialEnableTxFlow` --- Enables software flow control on transmit (XON/XOFF)
- `SerialEnableRxFlow` --- Enables software flow control on receive (XON/XOFF)
- `SerialEnableHwTxFlow` --- Enables hardware flow control (RTS/CTS)
- `SerialEnableHwRxFlow` --- Enables hardware flow control (RTS/CTS)
- `Serial485HalfDupMode` --- Toggles half-duplex mode for Serial RS-485
- `SendBreak` --- Sets a break in the transmission for a given period of time
- `serwriteaddress` --- Writes address values (Note: For multidrop parity mode only)

#### For the SB72, SB70, NBPK70, and Mod5270 Platforms

- `GetUartErrorReg` --- Gets the UART error register

#### For the SB72EX Platform

- `GetCD` --- Returns the state of the Data Carrier Detect (CD) pin of the Serial port
- `GetRI` --- Returns the state of the Ring Indicator (RI) pin of the Serial port
- `GetDSR` --- Returns the state of the Data Set Ready (DSR) pin of the Serial port
- `SetDTR` --- Returns the state of the Data Terminal Ready (DTR) pin of the Serial port

## 16.2. OpenSerial

### Synopsis:

```
int OpenSerial( int portnum, unsigned int baudrate, int stop_bits, int
data_bits, parity_mode parity );
```

### Description:

This function opens one of the UART Serial ports on your NetBurner device for reading and writing. The Serial port to be opened is provided by the input parameter portnum followed by the configuration parameters (i.e. baudrate, number of stop bits, number of data bits, and parity mode). The value returned after execution depends on whether or not the attempt was successful, or the type of error it caused (if it failed).

### Parameters:

| Type         | Name      | Description  |
|--------------|-----------|--|
| int          | portnum   | Determines which Serial port will be opened. The port number <b>must</b> be 0, 1, or 2. <b>Note:</b> Port 0 is the primary Serial port.                      |
| unsigned int | baudrate  | Specifies the baudrate of the opened Serial port. <b>Note:</b> This value <b>must</b> be: 300, 600, 1200, 2400, 4800, 9600, 119200, 38400, 57600, or 115200. |
| int          | stop_bits | Specifies the number of stop bits that the Serial port will use. <b>Note:</b> This value <b>must</b> be either 1 or 2.                                       |
| int          | data_bits | Specifies the number of data bits that the Serial port will use. <b>Note:</b> This value <b>must</b> be either 5, 6, 7, or 8.                                |
| parity_mode  | parity    | One of the four parity modes for the Serial port: eParityNone, eParityOdd, eParityEven, or eParityMulti.   |

### Returns:

The file descriptor (fd) of the specified Serial port number is returned upon successful opening of a port.

A negative number on error:

- SERIAL\_ERR\_PARAM\_ERROR (-4) --- Specified parameter for stop bits or data bits contains an invalid value.
- SERIAL\_ERR\_PORT\_ALREADYOPEN (-3) --- Specified Serial port number is already open.
- SERIAL\_ERR\_NOSUCH\_PORT (-1) --- Specified Serial port number does not exist; invalid port number value.

## 16.3. SimpleOpenSerial

### Synopsis:

```
SimpleOpenSerial( port, baud )
```

### Description:

This Macro opens one of the Serial ports (**using default values for stop bits, data bits, and parity**). For example, if you want to open Serial port 0 at 115200 baud, you could use the OpenSerial function:

```
int fd=OpenSerial( 0, 115200, 2, 8, eParityNone );
```

In reality, the stop bits are usually 2, the data bits are usually 8, and there is no parity. Therefore, to open Serial port 0 at 115200 baud, you would use this macro as follows:

```
SimpleOpenSerial( 0, 115200 );
```

**Note:** In the above example the "stop bits" value is set to 2, the "data bits" value is set to 8, and there is no parity. However, if you need to use a different value for any of the defaults (i.e. stop bits, data bits, and/or parity); you **must** use the OpenSerial function.

### Parameters:

| Type         | Name | Description  |
|--------------|------|--|
| int          | port | Determines which Serial port will be opened. The port number <b>must</b> be 0, 1, or 2. <b>Note:</b> Port 0 is the primary Serial port.                      |
| unsigned int | baud | Specifies the baudrate of the opened Serial port. <b>Note:</b> This value <b>must</b> be: 300, 600, 1200, 2400, 4800, 9600, 119200, 38400, 57600, or 115200. |

## 16.4. SerialClose

### Synopsis:

```
int SerialClose( int portnum );
```

### Description:

This function closes a UART Serial port. **Important:** It is valid to close a port that is **not** open. The port to be **closed** is specified by the input parameter **portnum**. The value returned depends on whether the execution was successful (or not), **or** the type of error it caused - if it failed.

This function is included to allow a port in an unknown state to be closed. Port settings can be changed by closing and reopening the port. **Note:** The close function also works.

### Parameter:

| Type | Name    | Description  |
|------|---------|--|
| int  | portnum | Specifies the Serial port to be closed. The port number <b>must</b> be 0, 1, or 2. <b>Note:</b> Port 0 is the primary Serial port. |

### Returns:

A return value of 0 (zero) confirms that the closing of the specified Serial port was successful.

A resource specific error code on failure:

- SERIAL\_ERR\_PORT\_NOTOPEN (-2) --- Specified Serial port number is already closed.
- SERIAL\_ERR\_NOSUCH\_PORT (-1) --- Specified Serial port number does not exist; (invalid port number value).



## 16.5. SerialEnableTxFlow

### Synopsis:

```
void SerialEnableTxFlow( int port, int enab );
```

### Description:

This function enables or disables (XON/XOFF) software flow control for the specified Serial port number.

### Parameters:

| Type | Name | Description  |
|------|------|--|
| int  | port | Specifies which Serial port will have <b>transmitter</b> flow control enabled (or disabled). |
| int  | enab | Enables (or disables) <b>software</b> flow control on the selected Serial port.              |

### Returns:

Nothing --- This is a void function.

## 16.6. SerialEnableRxFlow

### Synopsis:

```
void SerialEnableRxFlow( int port, int enab );
```

### Description:

This function enables or disables (XON/XOFF) software flow control for the specified Serial port number.

### Parameters:

| Type | Name | Description   |
|------|------|---|
| int  | port | Specifies which Serial port will have <b>receiver</b> flow control enabled (or disabled). |
| int  | enab | Enables (or disables) <b>software</b> flow control on the selected Serial port.           |

### Returns:

Nothing --- This is a void function

## 16.7. SerialEnableHwTxFlow

### Synopsis:

```
void SerialEnableHwTxFlow( int port, int enab );
```

### Description:

This function enables or disables transmitter hardware flow control for the specified serial port number. The transmitter is throttled via the CTS (Clear-to-Send) input signal line, which would be linked to the RTS (Request-to-Send) output line on the receiver. The transmitter starts sending data to the receiver when it receives a request on its CTS line from the receiver.

### Parameters:

| Type | Name | Description  |
|------|------|--|
| int  | port | Specifies which Serial port will have <b>transmitter</b> flow control enabled (or disabled). |
| int  | enab | Enables (or disables) <b>hardware</b> flow control on the selected Serial port.              |

### Returns:

Nothing --- This is a void function

## 16.8. SerialEnableHwRxFlow

### Synopsis:

```
void SerialEnableHwRxFlow( int port, int enab );
```

### Description:

This function enables or disables receiver hardware flow control for the specified serial port number. The receiver controls the flow of incoming data via the RTS (Request-to-Send) output signal line, which would be linked to the CTS (Clear-to-Send) input line on the transmitter. The receiver sets RTS when it is ready to receive data, thereby enabling the transmitter to start sending information to the receiver.

### Parameters:

| Type | Name | Description   |
|------|------|---|
| int  | port | Specifies which Serial port will have <b>receiver</b> flow control enabled (or disabled). |
| int  | enab | Enables (or disables) <b>hardware</b> flow control on the selected Serial port.           |

### Returns:

Nothing --- This is a void function

## 16.9. Serial485HalfDupMode

### Synopsis:

```
void Serial485HalfDupMode( int port, int enab );
```

### Description:

This function enables or disables half-duplex mode on the UART Serial RS-485 transmitter of the specified port number. Important: When half-duplex is disabled, full-duplex mode is enabled and vice versa.

### Parameters:

| Type | Name | Description  |
|------|------|--|
| int  | port | Specifies whether <b>Serial port 0</b> will have <b>half-duplex</b> mode enabled (or disabled).      |
| int  | enab | Enables (or disables) half-duplex mode.<br><b>Note:</b> When disabled - full-duplex mode is enabled. |

### Returns:

Nothing --- This is a void function.

## 16.10. SendBreak

### Synopsis:

```
void SendBreak( int port, DWORD time );
```

### Description:

This function sets a break in the UART transmission of the specified Serial port number for an amount of time indicated by the input parameter "time".

### Parameters:

| Type  | Name | Description   |
|-------|------|---|
| int   | port | Specifies which Serial port will have a break.  |
| DWORD | time | The unit of time used is the number of ticks, where one tick is equal to 1/20 second. |

### Returns:

Nothing --- This is a void function

## 16.11. serwriteaddress

### Synopsis:

```
int serwriteaddress( int fd, const char c );
```

### Description:

This function writes address characters provided by the input parameter "c" to the port number associated with the specified file descriptor (fd).

### Parameters:

| Type       | Name | Description  |
|------------|------|--|
| int        | fd   | Specifies the file descriptor of the Serial port where the address character will be sent. |
| const char | c    | The character to be written out with the ninth bit set.                                    |

### Returns:

(1) --- The serial address write operation was a success.

A negative number on error:

- SERIAL\_ERR\_PORT\_NOTOPEN (-2) --- Specified file descriptor has an associated Serial port that is closed; unable to transmit.
- SERIAL\_ERR\_NOSUCH\_PORT (-1) --- Transmission cannot take place because the associated Serial port does not exist; the file descriptor is invalid.

## 16.12. GetUartErrorReg

### Synopsis:

```
int GetUartErrorReg( int fd );
```

### Description:

This function (**SB72, SB70, NBPk70, and Mod5270 Platforms only**) gets the UART error register of the file descriptor associated with its Serial port. Any errors are added to the Serial port's error register in a logical "OR" operation. **Note:** Calling this function to read the error status **will** clear the register. The meaning of each bit in the error status register is shown in the table below.

| Bit | Meaning        |
|-----|----------------|
| 3   | Received break |
| 2   | Framing error  |
| 1   | Parity error   |
| 0   | Overflow error |

### Parameter:

| Type | Name | Description   |
|------|------|---|
| int  | fd   | Specifies the file descriptor of the Serial port number that will have its error register read. |

### Returns:

A positive number --- Successful execution (the read error status register is returned)

A negative number --- On error:

- SERIAL\_ERR\_PORT\_NOTOPEN (-2) --- Specified file descriptor has an associated Serial port that is closed.
- SERIAL\_ERR\_NOSUCH\_PORT (-1) ---The associated Serial port does not exist; the file descriptor is invalid.



## 16.13. GetCD

### Synopsis:

```
BOOL GetCD( int port );
```

### Description:

This function (SB72EX platform only) retrieves the current state of the CD (Data Carrier Detect) pin from the Serial port specified.

### Parameter:

| Type | Name | Description                       |
|------|------|-----------------------------------|
| int  | port | Specifies the Serial port number. |

### Returns:

True --- If CD (Data Carrier Detect) is set  
False --- If CD (Data Carrier Detect) is cleared

## 16.14. GetRI

### Synopsis:

```
BOOL GetRI( int port );
```

### Description:

This function (SB72EX platform only) retrieves the current state of the RI (Ring Indicator) pin from the Serial port specified.

### Parameter:

| Type | Name | Description                       |
|------|------|-----------------------------------|
| int  | port | Specifies the Serial port number. |

### Returns:

True --- If RI (Ring Indicator) is set  
False --- If RI (Ring Indicator) is cleared

## 16.15.     **GetDSR**

### **Synopsis:**

```
BOOL GetDSR( int port );
```

### **Description:**

This function (SB72EX platform only) retrieves the current state of the DSR (Data Set Ready) pin from the Serial port specified.

### **Parameter:**

| Type | Name | Description                       |
|------|------|-----------------------------------|
| int  | port | Specifies the Serial port number. |

### **Returns:**

True --- If DSR (Data Set Ready) is set

False --- If DSR (Data Set Ready) is cleared

## 16.16. SetDTR

### Synopsis:

```
void SetDTR( int port, BOOL val );
```

### Description:

This function (SB72EX platform only) sets the DTR (Data Terminal Ready) pin from the Serial port specified.

### Parameters:

| Type | Name | Description   |
|------|------|---|
| int  | port | Specifies the Serial port number.   |
| BOOL | val  | Setting this input parameter to <b>true</b> will <b>set</b> the DTR.<br>Setting this input parameter to <b>false</b> will <b>clear</b> the DTR. |

### Returns:

Nothing --- This is a void function

## 17. SNMP Library

### 17.1. Introduction

Simple Network Management Protocol (SNMP) is a system for exposing a number of variables to a Network Management System. These variables are grouped together into SNMP MIB's (Management Information Bases). It is common to underestimate the complexity and time required to implement SNMP for a product. The purpose of this section is to provide a general idea of the complexity and effort required to implement SNMP, and to describe the provisions of the NetBurner SNMP package.

**The NetBurner SNMP package is sold as a licensed option only, and is not part of the standard development kit package. Please contact our [Sales](#) Department to purchase the SNMP package.**

SNMP is complex, and the standard NetBurner SNMP package is intended for customers that have a SNMP expert on staff. If you do not know how to use and configure SNMP tools such as SNMPWALK, SNMPGET, and SNMPSET, you **must** acquire that capability **before** you attempt to implement SNMP on your system. If you do not know what a MIB is and do not know how to write one, you **must** acquire that capability **before** you attempt to implement a custom SNMP MIB. For additional information, please read the SNMP FAQ (<http://www.faqs.org/faqs/snmpfaq/>).

#### Implementation Requirements

- The Provisions of the NetBurner SNMP Package
- Items Not Included in the NetBurner SNMP Package
- Additional Support for SNMP

#### A Short Startup Guide for using NetBurner's SNMP Library

- Level 0 - Basic instructions on using the SNMP tools provided with the NetBurner SNMP package
- Level 1 - Enable SNMP at the absolute minimum level without custom MIBs
- Level 2 - A simple custom MIB to set/clear community names and trap destinations

#### Notes

- Note 1 - Custom tables (no external example code provided)
- Note 2 - Custom community name parsing and protection
- Note 3 - Traps and custom traps

#### Required Header File

```
#include <snmp.h>      // Found in C:\Nburn\include
```

#### Functions

- Snmpget - Get a single SNMP variable
- Snmpgetnext - Get the next variable following the specified variable
- Snmpset - Set a single SNMP variable
- Snmpwalk - Walk the SNMP tree

## 17.2. SNMP Implementation Requirements

### Provisions of the NetBurner SNMP Package

MIB-II (RFC 1213) implementation reports network usage variables from MIB-II. MIB-II is supported 100% with the following exceptions:

- The PPP interface does not support SNMP.
- The ipForwarding variable is read only and set to non-forwarding.
- The ipRouteTable - since the forwarding gateway is fixed; the route table is read only and set to a single value. The table will reflect the current state of the IP connections and routes stored in the ARP table, but it is not writable.
- The egp table and values are not applicable because NetBurner hardware is not a router and does not do egp.

### SNMP requires the maintenance and storage of a number of persistent variables

- The methods to set or modify variables such as SysLocation are defined in the MIB-II specification. The mechanisms to set or modify other variables (such as community names and trap destination IP addresses) are not defined and require the implementation of a custom mechanism. NetBurner provides a trivial NetBurner custom MIB to accomplish this task, but it should be part of your custom MIB. You will also have to write code that will take a structure containing this information and store it in nonvolatile memory so the names will be persistent (otherwise the SNMP community names will be hard coded to default values and traps will not be sent.).

### Tools for implementing a custom MIB on the NetBurner platform

- The tools provided will parse a custom MIB input file and produce a .CPP output file that implements the MIB. You **will** be responsible for hand editing this source file to populate the data in the MIB.

### Items NOT Included in the NetBurner SNMP Package

The NetBurner SNMP License Package explicitly does **not** provide:

1. Education on the SNMP protocol and how to use it
2. Technical support for writing your own custom MIB

### Additional Support for SNMP

NetBurner can provide this support on an hourly consulting basis. Given a group of experienced embedded developers with no SNMP knowledge or experience, the effort required to implement a custom SNMP MIB and learn to use the SNMP tools can easily exceed 100 hours of consulting time. In the extreme case of a complex implementation, it can exceed 300 hours. For more information on purchasing this support option, please contact our [Sales](#) Department.

## 17.3. Level 0 --- Basic Instructions Using the SNMP Tools

NetBurner provides a set of SNMP tools with its SNMP package. They are intended to assist in your SNMP development, but we consider them to be a convenience and not part of the core SNMP package. It is expected that anyone needing to develop SNMP application will already have some basic SNMP tools in-house. The four functions provided are:

- Snmpget --- Get a single SNMP variable
- Snmpgetnext --- GetNext a single SNMP variable
- Snmpset --- Set a single SNMP variable
- Snmpwalk --- Walk the SNMP tree

The SNMP protocol knows nothing about human readable names. The protocol only knows about variables identified by OIDs (Object Identifiers). For instance, the variable commonly referred to as sysDescr is really the OID 1.3.6.1.2.1.1.1.0.

The protocol on the wire will never know that 1.3.6.1.2.1.1.1.0 is usually called sysDescr. The SNMP tools do all of the translation from OID to human readable names by parsing MIB text files. Important: The NetBurner tools assume that these MIB's are stored in the C:\Nburn\mibs directory.

**Caution:** By convention, MIBS are **never** supposed to change. If you **change** the data stored in a MIB, then you are supposed to release a new MIB with a **different** OID. The meaning of a single OID and the contents of a specific MIB file, are **never** supposed to change. This is what the tools expect. This is completely unrealistic for a development environment, and hence developers typically use the following workaround: If you ever **change** the contents in **any** of the textual MIB files in the **C:\Nburn\mibs** directory, you **must** erase the file **.index** as that keeps a parsed and cached copy of the MIB information. Otherwise, the changes you made **will** seem to have **no** effect.

## 17.4. Level 1 --- Enable SNMP at the Absolute Minimum Level without Custom MIBs

To enable SNMP at the absolute **minimum** level (without custom MIBs), you **need** to do two things:

- Provide system identification information
- Provide storage and recall for the SYSInfo structure

You will find an example of this in your C:\Nburn\examples\snmp\simplesnmp directory.

### Providing System Identification Information

Every model of SNMP device is supposed to have a unique system identifier or OID. It is also supposed to have a human readable name. These are reported in the SNMP MIB-II standard variables sysDescr and sysObjectID. To define these constants you must define the following two variables in your application:

```
const char * SYSDESC="NetBurner SNMP Test application";

const char * SYSOID="1.3.6.1.4.1.8174.2.40";
```

The number sequence 1.3.6.1.4.1 is the SNMP tree pointing to the custom MIB area. **Warning:** The number sequence **8174** is the Private Enterprise Number uniquely issued to NetBurner. You should create a SYSOID that starts with 1.3.6.1.4 followed by your own company identifier. You can obtain an identifier from the Internet Assigned Numbers Authority (<http://www.iana.org/protocols/forms.htm>).

### Providing Storage and Recall for the SYSInfo Structure

The SNMP system needs to store and recall information that will be nonvolatile. This will usually be done by storing data in the UserFlash area. However, this was not done by default, as it is expected that the Users will probably be using this data area for their own storage structures. This SNMP storage is encapsulated in a SysInfo structure defined in snmp.h (found in C:\Nburn\include):

```
struct SysInfo
{
    char SysContact[256];
    char SysName[256];
    char SysLocation[256];
    unsigned char ReadCommunity[40];
    unsigned char WriteCommunity[40];
    IPADDR trap_destination;
    DWORD trap_enable_flags;
    DWORD valid;
};
```

This data is accessed and stored using two **user written** functions:



```
const SysInfo * GetSysInfo( );    /* Returns a SysInfo structure */  
  
void SetSysInfo( SysInfo si );    /* Saves a SysInfo structure */
```

**Note:** Simple examples of these functions can be found in C:\Nburn\examples\snmp\main.cpp. If this structure has never been initialized it is suggested that SysContact, SysName, and SysLocation default to either the empty string or to Not Set. The two community names should default to what ever you want your default read and write SNMP community names to be. (Think of community names as passwords.)

## 17.5. Level 2 --- A Simple Custom MIB to Set/Clear Community Names

SNMP specifies that community names are to be used for access control, but it does **not** specify how they are to be changed. The typical solution is to change them as part of your custom MIB. The example below will create an absolutely trivial custom MIB, and implement the ability to change the community name settings using this MIB. You **will** find a completed version of this project in your C:\Nburn\examples\snmp\nburnmib directory. There are five parts to this process.

### 1. Writing your Custom MIB

For this example we are going to implement the **absolutely minimal MIB**. You can copy this text into a text file. We will assume that this is named **NburnCnameMib.txt** for the purposes of this discussion. The full text is shown below.

#### Beginning of MIB file

```
NBURNSAMPLE-MIB DEFINITIONS ::= BEGIN
IMPORTS
    mgmt, enterprises, IPAddress
        FROM RFC1155-SMI
    OBJECT-TYPE
        FROM RFC-1212;
netburner OBJECT IDENTIFIER ::= { enterprises 8174 }
READCOMMUNITY OBJECT-TYPE
SYNTAX OCTET STRING (SIZE(1..255))
ACCESS read-write
STATUS mandatory
DESCRIPTION
"Description: ReadCommunity name"
::= {netburner 1}
WRITECOMMUNITY OBJECT-TYPE
SYNTAX OCTET STRING (SIZE(1..255))
ACCESS read-write
STATUS mandatory
DESCRIPTION
"Description: ReadCommunity name"
::= {netburner 2}
TRAPDESTINATION OBJECT-TYPE
SYNTAX IPAddress
ACCESS read-write
STATUS mandatory
DESCRIPTION
"Description: The Trap destination IP address"
::= {netburner 3 }
END
```

#### End of MIB file

## 2. Compiling your Custom MIB

The NetBurner SNMP tools provide a utility **SNMPTRANSLATE** that will convert a custom MIB into a cpp source file to implement the MIB. Important: The MIB **requires** several inputs, specifically RFC1155-SMI and RFC-1212. Just **copy** these files from the C:\Nburn\mibs directory to the directory where your **custom MIB** is located. To do this (copy) from the command line:

```
snmptranslate -M yourmibdir -Tn > yourmibfile.cpp
```

Therefore, the example below generates the cpp file that will be the new custom MIB.

```
snmptranslate -M ./ -Tn > Nburn_Cname_Mib.cpp
```

## 3. Implementing your Custom MIB Functions

This auto generated file only has the outline for what you want to do, you **will** need to go everywhere there is a **#warning** in the file and **add** your custom code for implementing the actual SNMP variables. In this case there are six functions to fill in:

```
/* Read function prototypes */
snmp_typeString ReadFuncREADCOMMUNITY();
snmp_typeString ReadFuncWRITECOMMUNITY();
snmp_typeIpAddress ReadFuncTRAPDESTINATION();

/* Write function prototypes */
int WriteFuncREADCOMMUNITY(snmp_typeString var, int bTest);
int WriteFuncWRITECOMMUNITY(snmp_typeString var, int bTest);
int WriteFuncTRAPDESTINATION(snmp_typeIpAddress var, int bTest);
```

We want these functions to report and set the variables we have defined. The bodies of these functions can be found in C:\Nburn\examples\snmp\Nburn\_Cname\_Mib.cpp

**Special Note:** We normally allow you to read SNMP variables with the read community name, and set them with the write community name. (Think passwords.) However, it would be a bit stupid if we could read the value of the write community name using only a READCOMMUNITY name; so one additional modification is made to the auto generated cpp file. We **must** change the read permissions of the WRITECOMMUNITY variable **from** READ\_COMMUNITY\_MASK **to** WRITE\_COMMUNITY\_MASK

## 4. Add your Custom MIB file to your Project

Now edit your makefile to add the mib cpp file (adding Nburn\_Cname\_Mib.cpp in the example case.).

## 5. Add your Custom MIB File to the MIB Tools

The MIB text file you created must be located where the MIB tools can find it. Copy the MIB text file you just created to your C:\Nburn\mibs directory, and **delete** the **.index** file (so a new index will automatically be created by the MIB tools). Your new MIB is now ready to be used.

## 17.6. Note 1 --- Custom Tables

### Implementing Tables

The general process for implementing your own custom MIB closely follows the steps outlined in the Level 2. One area that is somewhat different is tables. If you do **not** already know what an SNMP table is, you **need** to research the topic before reading this section. When the SNMP translate utility parses a MIB definition for a table, it generates a different set of functions. This **will** be illustrated with the udpEntry table from MIB-II.

```
/* Function definitions for udpEntry */
void AddTableElementudpEntry( void * data_el, snmp_typeIpAddress
udpLocalAddress, snmp_typeINTEGER, udpLocalPort );
void RemoveTableElementudpEntry( void * data_el );
void PutTableElementsudpEntry( SNMP_Request & req, void * data_el, int
subid );
```

These three functions allow you to do three things:

1. Add a table entry/element
2. Remove a table entry/element
3. Provide the values for a table entry/element

The first two functions are completely written by the snmptranslate utility. The final function needs to be filled in by the programmer. The programmer has three responsibilities:

- Call AddTableElementudpEntry when a new UDP table element is created. This **must** include a **(void \*) data\_el** that encapsulates whatever data is needed to access this element.
- Call RemoveTableElementudpEntry when ever a UDP table element is to be destroyed. **Important:** This **must** include the same **(void \*) data\_el** passed in when the table was **created**.
- Fill in the code to convert the **(void \*) data\_el** into the specific MIB variables that make up the table element.

This function as generated by **SNMPTRANSLATE** is shown below:

```
void PutTableElementsudpEntry( SNMP_Request & req, void * data_el, int
subid )
{
switch(subid)
{
case 1: req.put_asn.PutIPAddr ( /* You must provide a conversion
from the data_el for udpLocalAddress */ );
break;
case 2: req.put_asn.PutInt ( /* You must provide a conversion from
the data_el for udpLocalPort */ );
break;
default: req.put_asn.PutNullObject();
}
}
```

The example on the previous page is implemented in the C:\Nburn\system\bcls.cpp and C:\Nburn\system\udp.cpp code set.

**Important:** At this time, the NetBurner SNMP system does **not** implement writing to dynamically created table elements. However, table elements can be created using the standard SNMP write variable definitions. If you have a specific need for dynamic writable tables, please contact NetBurner [Support](#) and we will assist you.

## 17.7. Note 2 --- Custom Community Name Parsing and Protection

The default NetBurner implementation provides **two** community names: read and write. It is often desirable to have multiple community names providing multiple levels of access and object visibility.

The NetBurner SNMP implementation can support **32 different access classes**. All visibility and access decisions are based on a **32 bit mask**. Each SNMP element includes a mask parameter. This is the last element in the variable definitions:

```
SNMPREADFUNC( sysDescr, "1.3.6.1.2.1.1.1.0", ASN_typeString,
ReadFuncsysDescr, READ_COMMUNITY_MASK );
```

The present code defines:

```
#define READ_COMMUNITY_MASK (0x0001)
#define WRITE_COMMUNITY_MASK (0x0002)
```

You could easily define an additional mask:

```
#define CUSTOM_COMMUNITY_MASK (0x0004)
```

To connect these mask values to the community name you would have to write a function to convert community names to mask values, and to place a pointer to that function in the function pointer:

```
DWORD (*SnmCommunityDecodeFunc) (const unsigned char * name);
```

### Example Code

```
DWORD MyCustomCommunityDecode( const unsigned char cname )
{
if ( strcmp ( cname,"MySecretW0rd" )==0 ) return
CUSTOM_COMMUNITY_MASK;
/* Otherwise return the default community name mask stuff */
return DefaultCommunityDecode( cname );
}
```

Then someplace in your system initialization you will need to setup the function pointer:

```
SnmCommunityDecodeFunc=MyCustomCommunityDecode;
```

## 17.8. Note 3 --- Traps and Custom Traps

The NetBurner SNMP system provides for three types of traps

- Auto generated traps within the SNMP system - The Authentication fail trap and warm start traps are auto generated within the SNMP system at the appropriate times. These traps are sent to the destination defined in the trap\_destination variable in the SysInfo structure (the same structure that maintains community names).
- Basic predefined traps without additional variables as generated by the application code - If you pass in a destination of 0; it uses the value stored in the SysInfo structure for trap\_destination. This is done with the following function:

```
SnmpBasicTrap( IPADDR dest, const char * community_name, int
generic_trap, int specific_trap );
```

- Custom traps with additional OID values attached - This feature uses the function:

```
void SnmpTrapWithData          /* As defined in snmp.h */
```

**Important:** This **requires** that you write a call-back function that **will** actually put the variables into the trap.

### Example Code

```
/* The callback function that fills in extra varbind defined in
your custom trap variables */
void TrapVarFunction(ASN * put_asn)
{
    put_asn->PutHeader(0x30); /* Var Bind */
    put_asn->PutOidFromString("1.3.6.1.4.1.8174.1");
    put_asn->PutOctetString("This is test message number 1");
    put_asn->FixUpHeader(); /* Var Bind */
    put_asn->PutHeader(0x30); /* Var Bind */
    put_asn->PutOidFromString("1.3.6.1.4.1.8174.2");
    put_asn->PutOctetString("This is test message number 2");
    put_asn->FixUpHeader(); /* Var Bind */
}
/* The function that actually sends the trap */
void SendTestTrap()
{
    SnmpTrapWithData(0, "public", SNMP_ENTERPRISE_TRAP, 1,
TrapVarFunction);
}
```

## 17.9. Snmpget

### Usage:

```
Snmpget device_address community_name object_name
```

### Description:

This function gets a single SNMP variable.

### Parameters:

| Name           | Description   |
|----------------|---|
| device_address | The device IP Address (e.g. 10.1.1.77) or DNS name.                                 |
| community_name | The community name used to access the device. (The rough equivalent of a password.) |
| object_name    | The textual or OID name of the object to be retrieved.                              |

### Examples:

```
Snmpget 10.1.1.77 public sysDescr.0  
Snmpget 10.1.1.77 public .1.3.1.2.1.1.1.0
```



## 17.10. Snmpgetnext

### Usage:

```
Snmpgetnext device_address community_name object_name
```

### Description:

This function gets the **next** variable **following** the **specified** variable. The entire MIB of a device could be obtained by executing a getnext on .1.3.1 followed by a getnext on each returned variable. Essentially, this would walk down the entire MIB tree the same way as Snmpwalk.

### Parameters:

| Name           | Description   |
|----------------|---|
| device_address | The device IP Address (e.g. 10.1.1.77) or DNS name.                                 |
| community_name | The community name used to access the device. (The rough equivalent of a password.) |
| object_name    | The textual or OID name of the object preceding the object to be retrieved.         |

### Examples:

```
Snmpgetnext 10.1.1.77 public sysDescr  
Snmpgetnext 10.1.1.77 public .1.3.1.2.1.1.1
```

## 17.11. Snmpset

### Usage:

```
Snmpset device_address community_name object_name object_type
```

### Description:

This function sets a single SNMP variable.

### Parameters:

| Name           | Description   |
|----------------|---|
| device_address | The device IP Address (e.g. 10.1.1.77) or DNS name.                                 |
| community_name | The community name used to access the device. (The rough equivalent of a password.) |
| object_name    | The textual or OID name of the object to be set.                                    |
| object_type    | The type of the object. (See the “Types” description below.)                        |

### Types:

One of: i, u, t, a, o, s, x, d, n

- i --- INTEGER
- u --- unsigned INTEGER
- t --- TIMETICKS
- a --- IPADDRESS
- o --- OBJID
- s --- STRING
- x --- HEX STRING
- d --- DECIMAL STRING
- U --- unsigned int64
- I --- signed int64
- F --- float
- D --- double

### Example:

```
Snmpset 10.1.1.77 public sysLocation.0 s "At NetBurner HQ"
```

## 17.12. Snmpwalk

### Usage:

```
Snmpwalk device_address community_name
```

### Description:

This function walks the SNMP tree. Essentially it does a repetitive getnext until it runs out of SNMP variables to retrieve.

### Parameters:

| Name           | Description   |
|----------------|---|
| device_address | The device IP Address (e.g. 10.1.1.77) or DNS name.                                 |
| community_name | The community name used to access the device. (The rough equivalent of a password.) |

### Example:

```
Snmpwalk 10.1.1.77 public
```

## 18. SSL Library

### 18.1. Introduction

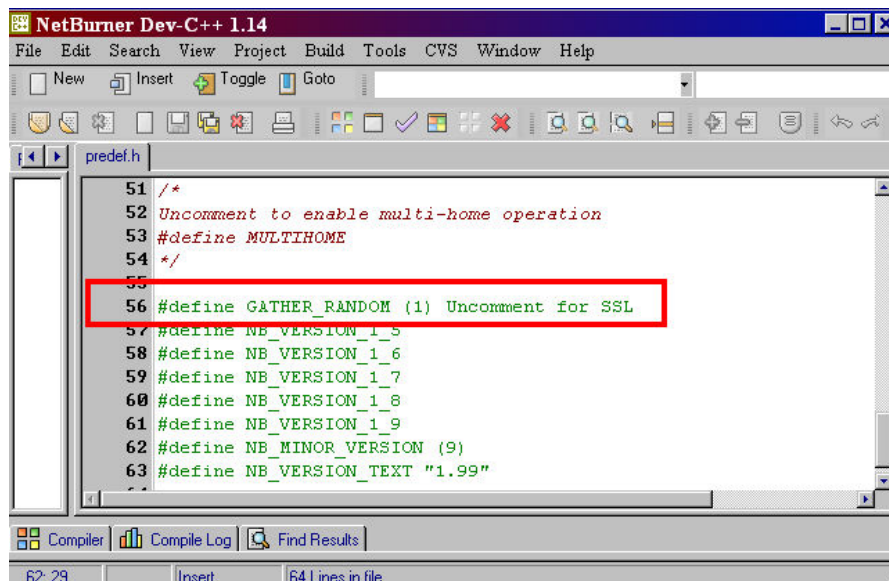
The NetBurner SSL package is sold as a licensed option only, and is not part of the standard development kit package. Please contact our [Sales](#) Department to purchase the SSL package.

Implementing SSL in an embedded system **will** require some knowledge of SSL certificates. Please read the following four SSL sections:

- Easy SSL overview
- Creating SSL server certificates
- Diagram: Creating a code module - SSL Server Key and Certificates
- Creating the list of acceptable client certificates

You **cannot** skip these documents. **Before** you can use the SSL accept function, you **will** need to add a Server certificate to your project. **Before** you can use the SSL connect function, you **will** need to add a list of Client certificates to your project.

**Important:** Before you compile any programs, open up **predef.h** (located in **C:\Nburn\include**) with any text editor, and **uncomment line 56** to get your applications to compile when using the NetBurner SSL Module. **After** editing **predef.h** (i.e. uncomment line 56), you **must** execute the **make clean** command (at the command line) in your **C:\Nburn\system** directory.



**Warning:** If you do not edit predef.h, your applications will not compile – you will get a make error as shown below (DOS Command line compilation).

```

cryptolib/random.cpp:44:2: #error GATHER_RANDOM must be defined uncomment it in nburn\include\p
predef.h
make[1]: *** [NetBurnerdepend.mk] Error 1
make[1]: Leaving directory `./nburn/system'
make: *** [/nburn/lib/NetBurner.a] Error 2

```

## Required Header File

```
#include <ssl.h>           // Found in C:\Nburn\include\crypto
```

## SSL Server Functions

- StartHTTPs --- Starts the secure Web Server
- SSL\_accept --- SSL mirror of the TCP accept function

## File Descriptor (fd) Information Functions

- IsSSLfd --- Is the file descriptor an SSL file descriptor or some other kind
- SSL\_GetSocketRemoteAddr --- Returns the remote address of this connected socket
- SSL\_GetSocketRemotePort --- Returns the remote port of this connected socket
- SSL\_GetSocketLocalAddr --- Returns the local address of this connected socket
- SSL\_GetSocketLocalPort --- Returns the local port of this connected socket

## Socket Option Functions

- SSL\_setsockopt --- Set the socket option
- SSL\_clrsockopt --- Clear the socket option
- SSL\_getsockopt --- Get the socket option

## SSL Client Function

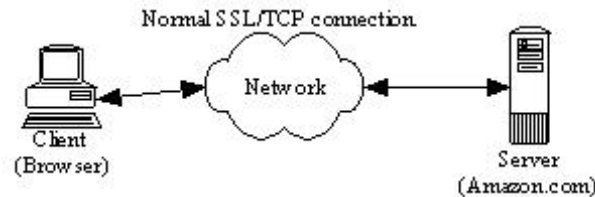
- SSL\_connect --- SSL mirror of the TCP connect call

## 18.2. SSL Overview

The NetBurner SSL library makes SSL as easy as it can be, but SSL **requires** a system of trusted certificates. The NetBurner SSL package is sold as a **licensed option** only, and is **not** part of the standard development kit package. Please contact our [Sales](#) Department if you want to purchase the SSL package.

When you use SSL to connect to <http://www.amazon.com> (for example) with a normal web browser, you will not need to know anything about certificates. This is because Amazon **purchased** a certificate from Verisign and your browser vendor **preinstalled** Verisign, as an entity that can sign **trusted** certificates.

If you know all there is to know about the whys and hows of certificates you can skip ahead to the section on installing certificates in your NetBurner based product. For the vast majority of the embedded developers, certificates will need a bit more explanation.



Above, is a picture of a perfectly normal TCP or SSL connection; the client (most often a browser) has connected through the network to a server. If we do **not** have any entities doing bad things on our network then there is **no** need for SSL. However, if the data we are sending is worth stealing, we might have a very different network picture (below).



If our connection is routed through a third party (a normal TCP connection), we have no guarantee that this third party is not a "bad guy" trying to steal or modify our data. The SSL protocol was designed to eliminate this man in the middle attack. SSL is designed not only to make sure that the data we send over the network is hidden from snooping eyes, but it is also designed to make sure we are connected to the proper server without any "bad guys" in the middle. **This verification is done with Public Key (PK) Cryptography and a hierarchy of trust.**

Why do we trust a doctor when we go to the emergency room? We trust the doctor we have never met because we trust the hospital to employ qualified doctors. The hospital vouches for his skills and we trust the hospital. His medical school also vouches for him by giving him a diploma with his name and the schools seal or signature. We trust the school, we trust the hospital, and thus we trust the doctor.

SSL works in a very similar way. When a client connects to the server the server sends the client a certificate. This certificate has three major elements:

1. A name (i.e. who is this server)
2. A public key (e.g. think of an open padlock)
3. A signature (by a trusted third party that vouches for the name and the public key)

A doctor's diploma is very similar; it also has three major elements: The doctor's name, the type of degree and the medical school (that vouches for the doctor). For example, Bob and George both graduate from Harvard Medical School. They both have Harvard diplomas. However, the diplomas are unique to each doctor. The diplomas are not interchangeable. Bob's diploma would be of no use to George and vice versa. This illustrates the first key point about SSL.

**Key Point # 1: Each and every SSL server must have a unique certificate. Note: You cannot reuse an SSL server certificate. The certificates are distinguished by the "common name" or "CN" on the certificate.**

If you went into a new doctor's office and saw a diploma from Harvard medical school, you would feel comfortable with the doctor's skills. You trust Harvard and Harvard will vouch for this doctor. If instead, the diploma were from the Medical School of Zaire, you would probably be more skeptical. We do not have the same inherent trust of this school as we did with Harvard.

In SSL as a client, we have to decide who we will trust to sign our certificates. This list of trusted certificate authorities must be explicitly configured into the client. When the web browser or OS was installed on your PC, it probably installed a list of trusted certificate authorities. With the NetBurner SSL library, we have to explicitly decide whom we are going to trust to sign server certificates. This leads to key point #2.

**Key Point # 2: An SSL client must be pre-configured with a list of Certificate Authorities (CAs) that it will trust to sign server certificates. This list can be common across all the clients and does not have to be unique.**

## What do I need to do to make SSL work?

- You **must** create or choose a certificate authority. **Note:** If you **create** a certificate authority, you **will** also have to create a set of public/private keys for this authority.
- You **must** create public/private keys and a certificate for **each** SSL server and have the certificate **signed** by the certificate authority you have chosen.
- You **must** configure the clients with the **list** of certificate authorities it should **trust**.

## How do I find or create a certificate authority?

Using the medical school example, you can go to Harvard pay lots of \$\$\$ and get a diploma that is trusted by everyone. You can also choose to start your own medical school and issue diplomas. Almost everyone in the world would trust a Harvard diploma. Almost no one will trust a "Bob's Medical School" diploma, unless you spend the time convincing them that it is a quality medical school. In the end, you will likely only be able to convince your family, and then only for non-life threatening needs.

SSL certificates are a lot like medical schools; you can go and purchase server certificates. To see what a certificate looks like, open your web browser (e.g. Internet Explorer), and connect to <https://www.NetBurner.com> (notice the s on the end of https). On Internet Explorer's menu, choose File then Properties. Now, click the Certificates button, and look at all the tabs shown in this section.

## How do I know whom my browser trusts?

On your (Internet Explorer) browser's menu - choose Tools then Internet Options. Open the Content tab, click the Certificates button, and open the Trusted Root Certificate Authorities tab. Add Verisign or Thawte and every browser in the world will trust your certificate and your server.

If you want to save some money and create your own certificate authority then you can do so. However, none of the clients will accept your certificate until you convince them to add "Bob's Certificate Authority" to their list of trusted certificate authorities. If, the users using the embedded SSL system you are deploying are all in one business entity, then it is relatively simple to add your own certificate authority to the list of trusted authorities. If you are responsible for both the client and server end of the connection, it is even easier; you can configure the clients to accept a single server authority - yours.

SSL is based on Public Key Cryptography (PK) and a little bit of background on PK is necessary in order to deploy a secure SSL solution. Public Key Cryptography is different from Symmetric Key Cryptography. In PK, the keys used for encryption are broken into two parts, much like a padlock (the public part) and a key (the private part). If you give someone an open padlock and a steel box, they can put things into the box, close the lid, and lock the lock. Unless they have the key to the lock, they cannot open the box. They can be confident that if they mail you the box, none of the mailmen along the way can look inside. Only the person who holds the (private) key to the padlock can open the box. For additional information on Public Key Cryptography, please read the Cryptography FAQ (<http://isc.faqs.org/faqs/cryptography-faq>)

When the SSL client connects to a server, the server sends back a certificate with a public key (open padlock). This certificate also includes the name of the server and a signature vouching for both the public key and the name. If any part of the certificate is changed, the signature will compute to be invalid.

So, if we have a "bad guy" in the middle, he can watch the padlock going from the server to the client. But, when the client puts his secret information into the box and locks it, the "bad guy" cannot see inside. He only knows that the client sent something in the box to the server. The secrets in the box are safe from the prying eyes of the "bad guy". This safety only exists if the server has done a good job of protecting the private key. If the "bad guy" sneaks into the server room, logs on the server console, and makes a copy of the private key, he can intercept all of the traffic. He can also change the content at will. This leads to key point #3.

**Key Point # 3: When using Public Key Cryptography (as SSL does), the system is only as secure as the security of the private key. Since a server needs access to the private key to unlock the data from the client, the private key must exist on the server.**

**Key Point # 3 Corollary: If the private key exists on the server, then the system is only as secure as the physical security of the server. If the server is not physically secure, then someone (i.e. the "bad guy") can attach an emulator or other hardware to the server and read out the private key.**

For example, suppose the "bad guy" wants to intercept your credit card number when you send it to Amazon to order a book. We have already shown that he cannot read the data unless he has Amazon's private key. However, he has one other option - he can pretend to be Amazon and offer his own certificate to you, the client. If this certificate is properly signed by a Certificate Authority the client trusts, then client will accept the connection. If any "Certificate Authority" in the list of trusted authorities is compromised, then the system is insecure. If the "bad guy" has the ability to add a new "Certificate Authority" to the client, then he can completely compromise the system. This leads to key point #4.



**Key Point # 4: If the ability to add a "Certificate Authority" to the client's list of trusted authorities is not secure, then system is not secure.**

**Key Point # 4 Corollary: If the list of trusted "Certificate Authorities" exists on the client, then the system is only as secure as the physical security of the client. If the client is not physically secure, then someone (i.e. the "bad guy") can attach an emulator or other hardware to the client and add a "trusted" authority.**

These last two key points imply that it is not possible to build a system that is more secure than the physical security of the device being secured. Important: All the cryptography in the world will not help if someone can gain access to your computer and hide a bug inside the keyboard; or even easier, add or modify a system file to record your keystrokes and periodically send them over the internet to some nefarious foe. Note: If your data is valuable enough to be attractive to a skilled adversary, then you must learn to be truly paranoid.

## Recommended Reading

For an excellent overview of computer security

- Secrets and Lies by Bruce Schneier (ISBN 0-471-25311-1)

For a detailed review of cryptography

- Applied Cryptography by Bruce Schneier (ISBN 0-471-11709-9)

For a detailed description of the SSL protocol

- SSL and TLS by Eric Rescorla (ISBN 0-201-61598-3)

For a reference on the math and methods in cryptography (this is a heavy duty book):

- Handbook of Applied Cryptography by Menezes, Oorschot and Vanstone (ISBN 0-8493-8523-7)

## 18.3. Creating a Code Module for SSL Server Certificates

### Introduction

The NetBurner SSL library provides some open source tools for the generation and maintenance of SSL keys and certificates. These key and certificate management tools are based on the fine **openssl package** available from <http://www.openssl.org>. These tools are subject to the openssl License. The **embedded SSL library code** is derived from other sources and is subject to the **standard NetBurner License** (located by default in **C:\Nburn\docs**)

### Setting up the Environment

**Important:** Before you can perform any of these steps, you **must** set up the environment.

- Make sure the **openssl.exe** provided with the NetBurner SSL Library is in your path (by default it installs in **C:\Nburn\pcbin**)
- Make sure you have **edited the openssl.cnf file in C:\Nburn\ssl\config** to have the appropriate entries for your business
- Make sure that your **system environment** has the variable **OPENSSL\_CONF= <your path >openssl.cnf**

### Creating a Certificate Authority (CA)

This step creates a CA you can use to sign SSL server certificates. **Important:** You should only have to do this step **once**. The key file created in this step should be **protected** as the security of **all** your SSL Certificates depends on it. You have **two** choices to protect this key file.

1. You can use a pass phrase to encode it
2. You can leave it unencoded to protect the computer it is stored on

**Note:** If you want the key to **not** be **protected** by a pass phrase then **leave** the **-des** off the **genrsa** command.

- Open a command prompt/DOS window
- Navigate to the directory you want to house your CA files
- To make a CA Key file, execute the command (and press the Enter key when finished):

```
openssl genrsa -out CA.key -des
```

- Create a CA Certificate, by executing the command (and press the Enter key when finished):

```
openssl req -new -key CA.key -x509 -days 3650 -out CA.crt
```

You **will** be prompted to answer some identification questions. How you answer them is up to you, but when creating a NetBurner CA we answered these questions as follows:

Country Name: US  
State or Province: California  
Locality: San Diego  
Organization Name: NetBurner  
Organizational Unit Name: Certificate Authority  
Common Name: NetBurner CA

If you are going to be accessing the embedded SSL device via a web browser you **will** need to add this Certificate to your web browser's list of trusted certificate authorities. To do this for Internet Explorer:

- Open up your Internet Explorer web browser
- From the Tools menu go into the Internet Options section
- Select the Content tab
- Press the Certificates button
- Select the Trusted Root Certification Authorities tab
- Press the Import button
- Select the CA.crt file to import. (**Note:** It will **not** show up via the browse button **unless** you **change** the file type combo box at the bottom of the window to look for **X509** certificate files.)

## Creating a Server Key

You will need to create a server key for each SSL Server you intend to deploy. If you are deploying many SSL servers, the bookkeeping associated with this will not be trivial.

- Open a command prompt/DOS window
- Navigate to the directory you want to house your device files
- To make a Device Key file, execute the command (and press the Enter key when finished):

```
openssl genrsa -out devicename.key
```

You **will** need to keep track of this key file while you make a server certificate, as the two have to be matched. If you are creating your own certificates, you can create a big batch file that does all of the steps in a single execution. See the batch file in Appendix I.

## Creating a Server Certificate with your CA

You will **need** to create a server certificate for **each** SSL Server you intend to deploy. If you are deploying many SSL Servers, the bookkeeping associated with this will not be trivial. The common name you enter in this step **must** match the deployed DNS name or the IP Address of the Server it will be used on.

- Open a command prompt/DOS window
- Navigate to the directory that you want to house your device files

- To make a Device Certificate Request file, execute the command (and press the Enter key when finished):

```
openssl req -new -key devicename.key -out devicename.csr
```

- To make a Device Certificate, execute the command (**all on one line**) and press the "Enter" key when finished:

```
openssl x509 -req -days 365 -in devicename.csr -CA CA.crt -CAkey CA.key  
-CAcreateserial -out device.crt
```

You can combine the creation of server keys, certificates, and code by running the batch file shown in Appendix I (at the end of this section).

## Converting a Certificate and Key to Code

This step takes both the private Server Key and the Server Certificate and converts them into a CPP source code module that can be linked into your application. This implies that you need to generate a different application image for each of your servers.

- Open a command prompt/DOS window
- Navigate to the directory that you want to house your device files
- To make a Device CPP file with the key in it, execute the command (and press the Enter key when finished):

```
openssl rsa -in devicename.key -nburn -out devicename.cpp
```

- To add the Certificate to the CPP file, execute the command (and press the Enter key when finished):

```
openssl x509 -nburn -in devicename.crt -append devicename.cpp
```

## Adding the Module to your Code Set

Take the devicename.cpp file previously created and add it to your makefile.

## Create a Server Certificate for External CA

If you are going to have your certificates signed by an external entity, they will **need** a Certificate Request file. **Note:** The common name you enter in this step **must** match the deployed DNS name or IP Address of the Server it will be used on.

- Open a command prompt/DOS window
- Navigate to the directory that you want to house your device files
- To make a Device Certificate Request file, execute the command (and press the Enter key when finished):

```
openssl req -new -key devicename.key -out devicename.csr
```

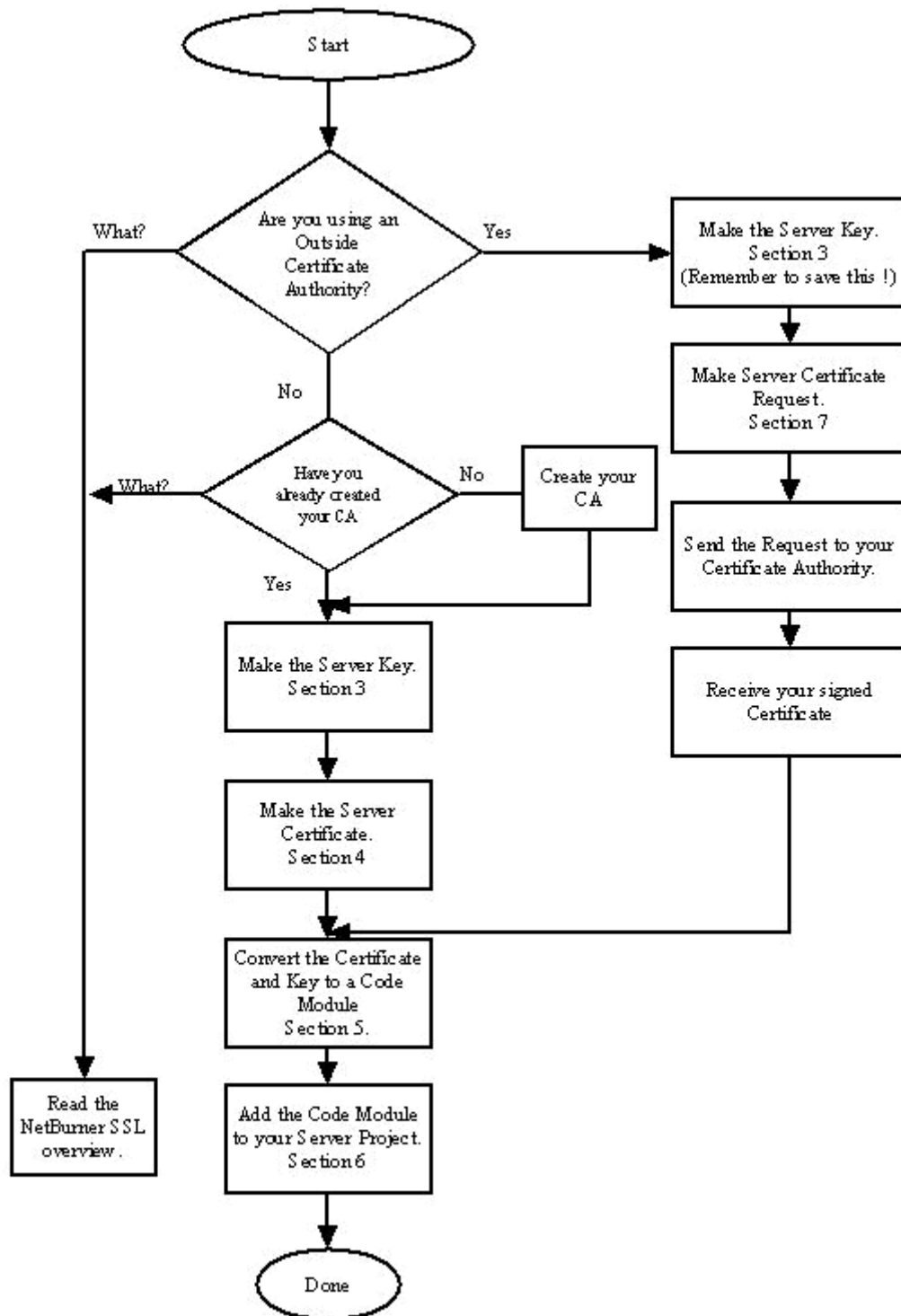
- Send this devicename.csr to the CA that will create your certificate.

**Warning:** If you lose the devicename.key file associated with this particular device, then you will not be able to use the certificate file they send back.

## Appendix I - A Device Creation DOS Batch File

```
REM contents of makedev.bat
REM usage is makedev devicename
openssl genrsa -out %1.key
openssl req -new -key %1.key -out %1.csr
openssl x509 -req -days 365 -in %1.csr -CA CA.crt -CAkey CA.key -
CAcreateserial -out %1.crt
openssl rsa -in %1.key -nburn -out %1.cpp
openssl x509 -nburn -in %1.crt -append %1.cpp
```

## 18.4. Creating a Code Module - SSL Server Key & Certificate - Diagram



## 18.5. Creating a Code Module for SSL Client Certificates

### Introduction

The NetBurner SSL library provides some open source tools for the generation and maintenance of SSL Keys and Certificates. These key and certificate management tools are based on the fine **openssl package** available from <http://www.openssl.org>. These tools are subject to the **openssl License**.

The **embedded SSL Library code** is derived from other sources and is subject to the standard **NetBurner license** (a copy is in your **C:\Nburn\docs** directory) that holds a list of certificate authorities that will be acceptable signers for SSL client connections.

### Determining what Certificates you need

If you do **not** know what certificates are needed you should **read** the **Easy SSL Overview** document in this section. You want to include the certificates from the CA(s) that **will** be signing the server certificates that your SSL client **will** be connecting to.

### Testing your Certificates

For each CA you want to trust you **need** to obtain a certificate. These certificates should be in the **X509 format** with the extension **.crt** (see RFC 3280). General purpose **openssl documents** can be found at: <http://www.openssl.org>. To test your certificate:

- Make sure the **openssl.exe** provided with the NetBurner SSL library is in your path (by default it installs in **C:\Nburn\pcbin**)
- Open a command prompt/DOS window in your project directory
- Execute the command (and press the Enter key when finished):

```
openssl x509 -in yourcert.crt -text
```

- This command should dump the cert contents
- If this command **fails** then you **will** need to **convert** the format. **Note:** One way to do this is to use the **openssl tools**
- Certificates can be in DER, NET and/or PEM formats. **Warning: We need the Certificates to be in the PEM format.** To convert a Certificate **from DER to PEM**, just execute the following command (and press the Enter key when finished):

```
openssl x509 -inform der -in server_cert.crt -out  
server_cert_in_pem.crt
```

## Creating a CA List file

This step creates a cpp file that holds all of the CA certificates you will accept. To create a CA List file:

- Open a command prompt/DOS window in your project directory
- Execute the command (and press the Enter key when finished):

```
openssl x509 -out ccerts.cpp -nburnccerts cerfile1 cerfile2  
...lastcertfile
```

- This command should build the file ccerts.cpp (Important: Make sure that this file (ccerts.cpp) has all of your certs in it.)
- Add this file to your project's makefile



## 18.6. StartHTTPs

### Synopsis:

```
void StartHTTPs( WORD ssl_port=443, WORD http_port=80 );
```

### Description:

This function starts the secure web server.

### Parameters:

| Type | Name         | Description                               |
|------|--------------|---|
| Word | ssl_port=443 | Port 443 is the standard HTTPS port.      |
| Word | http_port=80 | Port 80 is the standard HTTP Server port. |

### Returns:

Nothing --- This is a void function

## 18.7. SSL\_accept

### Synopsis:

```
int SSL_accept( int fdListen, IPADDR * address, WORD * port, WORD
timeout );
```

### Description:

This call is a mirror of the TCP accept call.

### Parameters:

| Type   | Name     | Description   |
|--------|----------|---|
| int    | fdListen | The file descriptor of the TCP listening socket.                    |
| IPADDR | *address | The IPADDR variable to hold the address of the connecting computer. |
| WORD   | *port    | The WORD variable to receive the remote port of this connection.    |
| WORD   | timeout  | The number of ticks to wait for a connection.                       |

### Return Values:

> 0 --- The file descriptor of the connected SSL socket  
TCP\_ERR\_TIMEOUT --- Underlying TCP system timed out  
TCP\_ERR\_NOCON --- The underlying TCP connection failed to negotiate  
TCP\_ERR\_CLOSING --- The underlying TCP fd was closing  
TCP\_ERR\_NOSUCH\_SOCKET --- The fd listen socket was invalid  
TCP\_ERR\_NONE\_AVAIL --- No free sockets to return  
TCP\_ERR\_CON\_RESET --- The connection was reset by the remote device  
TCP\_ERR\_CON\_ABORT --- The connection was aborted by the remote device  
SSL\_ERROR\_FAILED\_NEGOTIATION --- The SSL system failed to successfully negotiate a connection  
SSL\_ERROR\_HASH\_FAILED --- The connection failed the startup hash test  
SSL\_ERROR\_WRITE\_FAIL --- The connection failed to write out a full record

## 18.8. IsSSLfd

### Synopsis:

```
BOOL IsSSLfd( int fd );
```

### Description:

This Boolean function is used to determine if the fd (file descriptor) is an SSL connection (i.e. Is the file descriptor an SSL FD or some other kind?). It can be used by things like the callback functions of the web server to determine how secure the fd is before sending sensitive information over it.

### Parameter:

| Type | Name | Description                  |
|------|------|------------------------------|
| int  | fd   | The file descriptor to test. |

### Return Values:

TRUE --- If it is an SSL file descriptor  
FALSE --- If it is not an SSL file descriptor, or on error

## 18.9. SSL\_GetSocketRemoteAddr

### Synopsis:

```
IPADDR SSL_GetSocketRemoteAddr( int fd );
```

### Description:

This function returns the remote address of this connected socket. This function is used to retrieve the remote address of an SSL fd. This function will also work correctly if you pass in a fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

### Parameter:

| Type | Name | Description                  |
|------|------|------------------------------|
| int  | fd   | The file descriptor to test. |

### Return Values:

remote ---The IP Address of the TCP or SSL connection

0 --- Otherwise

## 18.10. SSL\_GetSocketRemotePort

### Synopsis:

```
WORD SSL_GetSocketRemotePort( int fd );
```

### Description:

This function returns the remote port of this connected socket. This function is used to retrieve the remote port of an SSL fd. This function will also work correctly if you pass in a fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

### Parameter:

| Type | Name | Description                  |
|------|------|------------------------------|
| int  | fd   | The file descriptor to test. |

### Return Values:

remote --- The port number of the TCP or SSL connection  
0 --- Otherwise

## 18.11. SSL\_GetSocketLocalAddr

### Synopsis:

```
IPADDR SSL_GetSocketLocalAddr( int fd );
```

### Description:

This function returns the local address of this connected socket. This function is used to retrieve the local address of an SSL fd. This function will also work correctly if you pass in a fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

### Parameter:

| Type | Name | Description                  |
|------|------|------------------------------|
| int  | fd   | The file descriptor to test. |

### Return Values:

local --- The IP address of the TCP or SSL connection  
0 --- Otherwise

## 18.12. SSL\_GetSocketLocalPort

### Synopsis:

```
WORD SSL_GetSocketLocalPort( int fd );
```

### Description:

This function returns the local port of this connected socket. This function is used to retrieve the local port of an SSL fd. This function will also work correctly if you pass in an fd that is a TCP connection. This allows you to use one set of code for both normal TCP and SSL connections.

### Parameter:

| Type | Name | Description                  |
|------|------|------------------------------|
| int  | fd   | The file descriptor to test. |

### Return Values:

local ---The port number of the TCP or SSL connection  
0 --- Otherwise

## 18.13. SSL\_setsockopt

### Synopsis:

```
int SSL_setsockopt( int fd, int option );
```

### Description:

This function will set the socket option.

### Parameters:

| Type | Name   | Description                  |
|------|--------|------------------------------|
| int  | fd     | The file descriptor to test. |
| int  | option | The socket option.           |

### Returns:

> 0 --- The file descriptor of the connected SSL socket  
TCP\_ERR\_TIMEOUT --- Underlying TCP system timed out  
TCP\_ERR\_NOCON --- The underlying TCP connection failed to negotiate  
TCP\_ERR\_CLOSING --- The underlying TCP fd was closing  
TCP\_ERR\_NOSUCH\_SOCKET --- The fd listen socket was invalid  
TCP\_ERR\_NONE\_AVAIL --- No free sockets to return  
TCP\_ERR\_CON\_RESET --- The connection was reset by the remote device  
TCP\_ERR\_CON\_ABORT --- The connection was aborted by the remote device  
SSL\_ERROR\_FAILED\_NEGOTIATION --- The SSL system failed to successfully negotiate a connection  
SSL\_ERROR\_HASH\_FAILED --- The connection failed the startup hash test  
SSL\_ERROR\_WRITE\_FAIL --- The connection failed to write out a full record  
SSL\_ERROR\_CERTIFICATE\_UNKNOWN SSL --- Received a certificate it could not decode  
SSL\_ERROR\_CERTIFICATE\_NAME\_FAILED The connected name did not match  
common\_name  
SSL\_ERROR\_CERTIFICATE\_VERIFY\_FAILED --- The server returned a certificate that we did not trust



## 18.14. SSL\_clrsocketoption

### Synopsis:

```
int SSL_clrsocketoption( int fd, int option );
```

### Description:

This function will clear the socket option.

### Parameters:

| Type | Name   | Description                  |
|------|--------|------------------------------|
| int  | fd     | The file descriptor to test. |
| int  | option | The socket option.           |

### Returns:

> 0 ---The file descriptor of the connected SSL socket  
TCP\_ERR\_TIMEOUT --- Underlying TCP system timed out  
TCP\_ERR\_NOCON --- The underlying TCP connection failed to negotiate  
TCP\_ERR\_CLOSING --- The underlying TCP fd was closing  
TCP\_ERR\_NOSUCH\_SOCKET --- The fd listen socket was invalid  
TCP\_ERR\_NONE\_AVAIL --- No free sockets to return  
TCP\_ERR\_CON\_RESET --- The connection was reset by the remote device  
TCP\_ERR\_CON\_ABORT --- The connection was aborted by the remote device  
SSL\_ERROR\_FAILED\_NEGOTIATION --- The SSL system failed to successfully negotiate a connection  
SSL\_ERROR\_HASH\_FAILED The connection failed the startup hash test  
SSL\_ERROR\_WRITE\_FAIL The connection failed to write out a full record  
SSL\_ERROR\_CERTIFICATE\_UNKNOWN SSL --- Received a certificate it could not decode  
SSL\_ERROR\_CERTIFICATE\_NAME\_FAILED --- The connected name did not match common\_name  
SSL\_ERROR\_CERTIFICATE\_VERIFY\_FAILED --- The server returned a certificate that we did not trust.

## 18.15. SSL\_getsockopt

### Synopsis:

```
int SSL_getsockopt( int fd );
```

### Description:

This function will get the socket option.

### Parameter:

| Type | Name | Description                  |
|------|------|------------------------------|
| int  | fd   | The file descriptor to test. |

### Returns:

> 0 --- The file descriptor of the connected SSL socket  
TCP\_ERR\_TIMEOUT --- Underlying TCP system timed out  
TCP\_ERR\_NOCON --- The underlying TCP connection failed to negotiate  
TCP\_ERR\_CLOSING --- The underlying TCP fd was closing  
TCP\_ERR\_NOSUCH\_SOCKET --- The fd listen socket was invalid  
TCP\_ERR\_NONE\_AVAIL --- No free sockets to return  
TCP\_ERR\_CON\_RESET --- The connection was reset by the remote device  
TCP\_ERR\_CON\_ABORT --- The connection was aborted by the remote device  
SSL\_ERROR\_FAILED\_NEGOTIATION --- The SSL system failed to successfully negotiate a connection  
SSL\_ERROR\_HASH\_FAILED --- The connection failed the startup hash test  
SSL\_ERROR\_WRITE\_FAIL --- The connection failed to write out a full record  
SSL\_ERROR\_CERTIFICATE\_UNKNOWN\_SSL --- Received a certificate it could not decode  
SSL\_ERROR\_CERTIFICATE\_NAME\_FAILED --- The connected name did not match common\_name  
SSL\_ERROR\_CERTIFICATE\_VERIFY\_FAILED --- The server returned a certificate that we did not trust

## 18.16. SSL\_connect

### Synopsis:

```
int SSL_connect( IPADDR ip, WORD local_port, WORD remote_port, DWORD
timeout, const char * common_name );
```

### Description:

This call is a mirror of the TCP connect call.

### Parameters:

| Type       | Name         | Description   |
|------------|--------------|---|
| IPADDR     | ip           | The address to connect to.  |
| WORD       | local_port   | The local port to use. <b>Note:</b> 0 will pick a local port.   |
| WORD       | remote_port  | The port to connect to.   |
| DWORD      | timeout      | The number of ticks to wait for a connection.   |
| const char | *common_name | The command name to use for checking certificate validity. <b>Note:</b> Passing in NULL will accept any connection. |

### Return Values:

> 0 ---The file descriptor of the connected SSL socket  
TCP\_ERR\_TIMEOUT --- Underlying TCP system timed out  
TCP\_ERR\_NOCON --- The underlying TCP connection failed to negotiate  
TCP\_ERR\_CLOSING --- The underlying TCP fd was closing  
TCP\_ERR\_NOSUCH\_SOCKET --- The fd listen socket was invalid  
TCP\_ERR\_NONE\_AVAIL --- No free sockets to return  
TCP\_ERR\_CON\_RESET --- The connection was reset by the remote device  
TCP\_ERR\_CON\_ABORT --- The connection was aborted by the remote device  
SSL\_ERROR\_FAILED\_NEGOTIATION --- The SSL system failed to successfully negotiate a connection  
SSL\_ERROR\_HASH\_FAILED --- The connection failed the startup hash test  
SSL\_ERROR\_WRITE\_FAIL --- The connection failed to write out a full record  
SSL\_ERROR\_CERTIFICATE\_UNKNOWN SSL --- Received a certificate it could not decode  
SSL\_ERROR\_CERTIFICATE\_NAME\_FAILED --- The connected name did not match common\_name  
SSL\_ERROR\_CERTIFICATE\_VERIFY\_FAILED --- The server returned a certificate that we did not trust

## 19. Stream Update Library

### 19.1. Introduction

The functions supplied in this module are intended to be used in conjunction with FTP Server and FTP Client implementations. Functions are provided to access the user parameter storage area of the flash memory, and to update the application code in flash memory.

#### Required Header File

```
#include <StreamUpdate.h>           // Found in C:\Nburn\include
```

#### User Parameter Flash Data Functions

- `SendUserFlashToStreamAsBinary` --- Send User Parameter Flash data as a binary output stream
- `SendUserFlashToStreamAsS19` --- Send User Parameter Flash data as a S19 ASCII record to an output stream
- `ReadS19UserFlashFromStream` --- Read User Parameter Flash data from a S19 ASCII input stream
- `ReadBinaryUserFlashFromStream` --- Read User Parameter Flash data from a binary input stream

#### Application Code Function

- `ReadS19ApplicationCodeFromStream` --- Read a new application in `_APP.S19` format from an ASCII input stream

#### Example Application

- `ftpd_code_update` (found in `C:\Nburn\examples`)

## 19.2. SendUserFlashToStreamAsBinary

### Synopsis:

```
int SendUserFlashToStreamAsBinary( int fd );
```

### Description:

This function sends User Parameter Flash data as a binary output stream. This function sends the User Parameter Flash data to the specified fd output stream as a binary record.

### Parameter:

| Type | Name | Description  |
|------|------|--|
| int  | fd   | The socket file descriptor (fd) to send the data to. |

### Return Values:

STREAM\_UP\_OK --- The system was able to send the data

STREAM\_UP\_FAIL --- The system failed to send the data

## 19.3. SendUserFlashToStreamAsS19

### Synopsis:

```
int SendUserFlashToStreamAsS19( int fd );
```

### Description:

This function sends User Parameter Flash data as a S19 ASCII record to an output stream. This function sends the User Parameter Flash data to the specified fd output stream as a S19 text record.

### Parameter:

| Type | Name | Description  |
|------|------|--|
| int  | fd   | The socket file descriptor (fd) to send the data to. |

### Return Values:

STREAM\_UP\_OK --- The system was able to send the data

STREAM\_UP\_FAIL --- The system failed to send the data

## 19.4. ReadS19UserFlashFromStream

### Synopsis:

```
int ReadS19UserFlashFromStream( int fd );
```

### Description:

This function reads User Parameter Flash data from a S19 ASCII input stream. This function reads ASCII S19 records from the specified fd input stream and programs the data in the User Parameter Flash area.

### Parameter:

| Type | Name | Description  |
|------|------|--|
| int  | fd   | The socket file descriptor (fd) to read data from. |

### Return Values:

STREAM\_UP\_OK --- The system was able to read the data and update flash  
STREAM\_UP\_FAIL ---The system failed to read or update

## 19.5. ReadBinaryUserFlashFromStream

### Synopsis:

```
int ReadBinaryUserFlashFromStream( int fd );
```

### Description:

This function reads User Parameter Flash data from a binary input stream. This function reads binary data from the specified input stream and programs the data into the User Parameter Flash area.

### Parameter:

| Type | Name | Description  |
|------|------|--|
| int  | fd   | The socket file descriptor (fd) to read data from. |

### Return Values:

STREAM\_UP\_OK --- The system was able to read the data and update flash  
STREAM\_UP\_FAIL --- The system failed to read or update



## 19.6. ReadS19ApplicationCodeFromStream

### Synopsis:

```
int ReadS19ApplicationCodeFromStream( int fd );
```

### Description:

This function reads a new application in \_APP.S19 format from an ASCII input stream. This function reads ASCII S19 records from a \_APP.s19 format application file and reprograms the Flash memory with the new application. **Note:** The Flash memory will **not** be modified unless the **entire** application is received **without** error.

Since applications are run from RAM, your NetBurner device **must** be rebooted **before** the new application code becomes active. One way to accomplish a reboot is to use the **ForceReboot** function (located in the NetBurner System Library section of this manual).

**Note:** The items that you **will** need to clean up and/or close before a reboot are dependant on your particular application. At a minimum, you should clean up and/or close any FTP Client or Server sessions **before** calling this function.

### Parameter:

| Type | Name | Description  |
|------|------|--|
| int  | fd   | The socket file descriptor (fd) to read data from. |

### Return Values:

STREAM\_UP\_OK --- The system was able to read the data and update the Flash  
STREAM\_UP\_FAIL --- The system failed to read or update Flash

### Example Application:

ftpd\_code\_update --- Located (by default) in C:\Nburn\examples

## 20. System Library

### Required Header Files

```
#include <system.h>           // Found in C:\Nburn\include
#include <utils.h>             // Found in C:\Nburn\include
```

### System Constant

#### Required Header File

```
#include <constants.h>        // Found in C:\Nburn\include
```

- Constants --- System configuration constants

### Structure

- ConfigRecord --- The configuration storage structure

### Global Variables

- gConfigRec --- The global configuration record
- Secs --- Seconds since the board booted
- TimeTick --- Time ticks since the board booted

### Code Update Functions/Capabilities

- Code Update Overview
- EnableAutoUpdate
- UpdateShutdown Hook
- UpdatePassword Hook

### Configuration Functions

- UpdateConfigRecord --- Change the configuration record stored in Flash
- SetupDialog --- Change system configuration by prompting the user over stdio

### User Flash Parameter Functions

- SaveUserParameters --- Save a blob of data to Flash
- GetUserParameters --- Get a read only pointer to the user defined data blob stored in Flash

## LED and Switch Functions

- `putleds` --- Set the system board LEDs
- `getdipsw` --- Read the DIP switches on the board

## Utility I/O Functions

- `ShowData` --- Show a data block in ASCII and hex
- `ShowMac` --- Show a MAC address on stdio
- `outbyte` --- Output a byte on stdio
- `print` --- Output a zero terminated string
- `putnum` --- Output a hexadecimal number to stdio
- `AsciiToIp` --- Convert a dotted decimal IP string to an IP address
- `ShowIP` --- Show an IP address as dotted decimal on stdio
- `itoa` --- An integer to ASCII function

## Diagnostic Function

- `ShowCounters` --- Show all system counters on stdio

## Reboot Function

### Required Header File

```
#include <bsp.h> // Found in C:\Nburn\include
```

- `ForceReboot` --- Reboot your NetBurner device

## Ethernet Functions

### Required Header File

```
#include <ethernet.h> // Found in C:\Nburn\include
```

- `EtherLink` --- Reports the status of the Ethernet link
- `EtherSpeed100` --- Reports if the Ethernet link is operating at 100 MB
- `EtherDuplex` --- Reports if the Ethernet link is in Full Duplex mode
- `ManualEthernetConfig` --- Controls the speed and duplex of the Ethernet connection

## 20.1. Constants

### Synopsis:

These constants control the system configuration.

**Warning:** If you change these constants, you must rebuild the System directory.

### Interrupt Priorities:

- `#define TICK_IRQ_LEVEL (5)`
- `#define SERIAL_IRQ_LEVEL (4)`
- `#define SERIAL_VECTOR_BASE (64)`

### Time Related Functions:

- `#define TICKS_PER_SECOND (20)`
- `#define XTAL_FREQ 4000000`

### Ethernet Buffer Definitions:

- `#define ETHER_BUFFER_SIZE 1548`
- `#define ETH_MAX_SIZE (1500)`
- `#define ETH_MIN_SIZE (64)`
- `#define MAX_UDPDATA (ETHER_BUFFER_SIZE-(20+8+14))`

### uCOS OS Setting:

- `#define OS_MAX_TASKS 20      // Max number of system tasks`

### System Task Priorities:

- `#define MAIN_PRIO (50)`
- `#define HTTP_PRIO (45)`

Any User Tasks that call I/O should have a Priority Number higher than the TCP\_PRIO:

- `#define TCP_PRIO (40)`
- `#define IP_PRIO (39)`
- `#define ETHER_SEND_PRIO (38)`

**Stack Size Definitions:**

```
// If you add big structures in user functions you should
// increase the value of the USER_TASK_STK_SIZE
• #define USER_TASK_STK_SIZE (2048)
// If you add big structures in your dynamic HTML code
// you need to increase the value of HTTP_STK_SIZE
• #define HTTP_STK_SIZE (2048)
• #define IP_STK_SIZE (2048)
• #define TCP_STK_SIZE (2048)
• #define IDLE_STK_SIZE (2048)
• #define ETHER_SEND_STK_SIZE (2048)
```

**TCP Constants:**

```
• #define DEFAULT_TCP_MSS (512)
// See RFC 1122 for a 50msec tick 60 ticks=3 sec 4*15=60
• #define DEFAULT_TCP_RTTVAR ((TICKS_PER_SECOND*3)/4)
  o (Note: The 4 comes from Stevens Vol. 1 page 300)
// 75 seconds Min
• #define TCP_CONN_TO (75 * TICKS_PER_SECOND)
// 200 msec delayed ACK timer
• #define TCP_ACK_TICK_DLY (TICKS_PER_SECOND /5)
• #define DEFAULT_INITIAL_RTO (TICKS_PER_SECOND*3)
• #define TCP_MAX_RTO (64 * TICKS_PER_SECOND)
• #define TCP_MIN_RTO (1 * TICKS_PER_SECOND)
• #define TCP_2MSL_WAIT (60 * TICKS_PER_SECOND)
• #define MAX_TCP_RETRY (12)
• #define TCP_WRITE_TIMEOUT (TICKS_PER_SECOND*10)
// Store 3 segments max in tx and rx buffers
• #define TCP_BUFFER_SEGMENTS (3)
// 10 idle Seconds and a partially received request is abandoned
• #define HTTP_TIMEOUT (TICKS_PER_SECOND*10)
```

**FD Offset Values:**

```
• #define SERIAL_SOCKET_OFFSET (3)
• #define TCP_SOCKET_OFFSET (5)
• #define MAX_IP_ERRS 3
• #define BUFFER_POOL_SIZE (64)
• #define UDP_DISPATCH_SIZE (10)
• #define ARP_ENTRY_SIZE (256)
• #define TCP_SOCKET_STRUCTS (32)
• #define UDP_NETBURNERID_PORT (0x4E42) /*NB*/
• #define TFTP_RX_PORT (1414)
```

## 20.2. ConfigRecord

### Synopsis:

```
typedef struct
{
    unsigned long recordsize; /* The stored size of the struct*/
    unsigned long ip_Addr; /* The device IP Address */
    unsigned long ip_Mask; /* The IP Address Mask */
    unsigned long ip_GateWay; /* The address of the IP Gateway */
    unsigned long ip_TftpServer; /* The address of the TFTP server to load
data from for debugging */
    unsigned long baud_rate; /* The initial system baud rate */
    unsigned char wait_seconds; /* The number of seconds to wait before
booting */
    unsigned char bBoot_To_Application; /* True - if we boot to the
application, not the monitor */
    unsigned char bException_Action; /* What should we do when we have an
exception? */
    unsigned char m_FileName[80]; /* The file name of the TFTP file to
load */
    unsigned char mac_address[6]; /* The Ethernet MAC address */
    unsigned long ip_DNS_server; /* The DNS Server address */
    unsigned long m_Unused[7];
    unsigned short checksum; /* A Checksum for this structure */
} ConfigRecord;
```

### Description:

This structure is stored in the system FLASH and records **default** values for the system operation. **Note:** This structure can be **manipulated** with the **IP Setup** program and the **Debug monitor**.

## 20.3. gConfigRec

### Synopsis:

```
extern ConfigRecord gConfigRec;
```

### Description:

This is a **read only copy** of the system configuration record.

## 20.4. Secs

### Synopsis:

```
extern VDWORD Secs;
```

### Description:

The number of seconds since the device rebooted.



## 20.5. TimeTick

### Synopsis:

```
extern VDWORD TimeTick;
```

### Description:

The number of time ticks since the device booted. There are TICKS\_PER\_SECOND ticks in each second. **Note:** At the time this document was written, this value is 20.

## 20.6. Code Update Overview

### Description:

It is **not** necessary to access the serial port to update the application code on your NetBurner device. With AutoUpdate capability, it is possible to update the application code running on your NetBurner device from **any** computer (with the AutoUpdate application running) with network access to that NetBurner device.

**Important:** AutoUpdate **must** be enabled in your application **before** it can be used. This is done by calling the **EnableAutoUpdate** function (see the example code below).

**Note:** If you want the update to **automatically** shutdown the running application **before** it updates the application then you need to add a function to the **UpdateShutdown Hook**. If you want the update to be **password protected**, you must also install a password checking function in the **UpdatePassword Hook**.

### Example Code:

```
*
*
*
#include <autoupdate.h> /* Required for AutoUpdate capability */
*
*
*
int MyPasswordTest(const char * user, const char * pass)
{
    /* Check password and user here */
    if (/*passwordok*/) return 1;
    else return 0;
}
int MyShutdownTest(void )
{
    if (/* It is OK to shutdown */)
    {
        /* Do your shutdown processing here */
        return 1;
    }
    else
        return 0;
}
void UserMain(void * pd)
{
    update_authenticate_func=MyPasswordTest;
    update_shutdown_func=MyShutdownTest;
    EnableAutoUpdate(); /* Required for AutoUpdate capability */
}
```

## 20.7. Enable AutoUpdate

### Synopsis:

```
void EnableAutoUpdate( );
```

### Description:

Calling this function enables the remote network AutoUpdate function. **Important:** This functionality can **only** be turned **on**. However, if you **need** to turn the capability **on and off dynamically**, then you **will** need to use the **UpdateShutdown Hook** functionality.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 20.8. Update Shutdown Hook

### Synopsis:

```
extern int ( *update_shutdown_func )( void );
```

### Description:

By pointing this function pointer to a function that function **will** be called **before** AutoUpdate takes place. It also allows the function to **reject** any update attempt.

Your function **must** be of the form:

```
int YourShutdownTest(void)
{
    if (/* It is OK to shutdown */)
    {
        /* Do your shutdown processing here */
        return 1;
    }
    return 0; /* If you want to abort AutoUpdate */
}
```

## 20.9. Update Password Hook

### Synopsis:

```
extern int ( *update_authenticate_func )( const char *name, const char
*passwd );
```

### Description:

Pointing this function pointer at a password checking function **requires** that the updating program provide a **password** that this routine approves of.

Your password function **must** be of the form:

```
int MyPasswordTest(const char * user, const char * pass)
{
    /* Check password and user here */
    if (/* password ok */) return 1;
    else return 0;
}
```

You **assign** the function to the password hook by:

```
update_authenticate_func=MyPasswordTest;
```

## 20.10. UpdateConfigRecord

### Synopsis:

```
void UpdateConfigRecord( ConfigRecord *pNewRec );
```

### Description:

This updates the stored configuration record. It can be used to change system configurations settings. **Note:** The mac\_address **and** checksum values are **ignored** in the passed-in structure, and the proper system values are used **before** storing the record.

### Returns:

Nothing --- This is a void function

## 20.11. SetupDialog

### Synopsis:

```
void SetupDialog( );
```

### Description:

This function will cause an interactive exchange on stdio. **Note:** This exchange **will** allow the user to change **both** the IP Address and Baudrate values.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 20.12. SaveUserParameters

### Synopsis:

```
int SaveUserParameters( void * pCopyFrom, int len );
```

### Description:

This function stores up to 8K of arbitrary data in the user configuration space. The type and format of the stored data is entirely up to the individual developer. The system stores this as a blob, and provides no protection from uninitialized data. **Note:** The developer needs to **add** some **uninitialized** data protection to his stored structure.

### Parameters:

| Type | Name       | Description   |
|------|------------|---|
| void | *pCopyFrom | A pointer to the data to store.   |
| int  | len        | The length of the data to store. <b>Note:</b> This value <b>must</b> be less than or equal to 8192. |

### Returns:

0 (zero) --- For failure  
1 --- If successful



## 20.13.      **GetUserParameters**

### **Synopsis:**

```
void * GetUserParameters( );
```

### **Description:**

This function returns a pointer to the user parameter area. This area is intended for storage of non-volatile configuration parameters. The type and format of the stored data is entirely up to the individual developer. The system stores this as a blob, and provides no protection from uninitialized data. Note: The developer needs to add some uninitialized data protection to his stored structure.

### **Parameters:**

None

### **Returns:**

A read only pointer to the user parameter area

### **Example Application:**

FlashForm --- Found in C:\Nburn\examples

## 20.14.     **putleds**

### **Synopsis:**

```
void putleds( unsigned char b );
```

### **Description:**

This function sets the LEDs on and off.

- 0x00 = all OFF
- 0xFF = all ON
- 0x01 = LED 1
- 0x02 = LED 2
- 0x04 = LED 3
- 0x08 = LED 4
- 0x10 = LED 5
- 0x20 = LED 6
- 0x40 = LED 7
- 0x80 = LED 8

### **Returns:**

Nothing --- This is a void function

### **Example Application:**

TicTacToe --- Found in C:\Nburn\examples

## 20.15.     **getdipsw**

### **Synopsis:**

```
unsigned char getdipsw( );
```

### **Description:**

This function reads the dip switches.

### **Parameters:**

None

### **Returns:**

The bit of the DIP switches:

- SW1 = 1
- SW2 = 2
- SW3 = 4
- SW4 = 8

### **Example Application:**

TicTacToe --- Found in C:\Nburn\examples

## 20.16. ShowData

### Synopsis:

```
void ShowData( PBYTE fromptr, WORD len );
```

### Description:

This function dumps a block of memory to stdio. It displays this block as hexadecimal and ASCII where appropriate.

### Returns:

Nothing --- This is a void function

## 20.17. ShowMac

### Synopsis:

```
void ShowMac( MACADR * ma );
```

### Description:

This function displays the MAC Address on stdout.

### Returns:

Nothing --- This is a void function

## 20.18. outbyte

### Synopsis:

```
void outbyte( char c );
```

### Description:

This function outputs a single character on stdout. **Note:** This is a very low overhead call.

### Returns:

Nothing --- This is a void function

## 20.19.     **print**

### **Synopsis:**

```
void print( char * );
```

### **Description:**

The output of print is a NULL terminated string to stdout. **Note:** This is a much lower overhead call than printf.

### **Returns:**

Nothing --- This is a void function

## 20.20. putnum

### Synopsis:

```
void putnum( int i );
```

### Description:

The output of putnum is a hexadecimal number of stdout. **Note:** This is a much lower overhead call than printf.

### Returns:

Nothing --- This is a void function



## 20.21.     **AsciiToIp**

### **Synopsis:**

```
IPADDR AsciiToIp( char * p );
```

### **Description:**

This function converts an ASCII representation to an IP Address.

### **Returns:**

An IP Address

### **Example:**

```
AsciiToIp("10.1.1.12");
```

## 20.22. ShowIP

### Synopsis:

```
void ShowIP( IPADDR ia );
```

### Description:

This function displays the IP Address on stdout.

### Parameter:

| Type   | Name | Description                           |
|--------|------|---------------------------------------|
| IPADDR | ia   | the IP Address that will be displayed |

### Returns:

Nothing --- This is a void function

### Example:

```
ShowIP(0x0a01010c); // Sends "10.1.1.12" to stdio
```

## 20.23. itoa

### Synopsis:

```
char * itoa ( int value, char * buffer, int radix )
```

### Description:

This function converts an integer value to a null-terminated string using the specified radix and stores the result in the given buffer. If the radix is 10 and the value is negative, the string is preceded by the minus sign (-). With any other radix, the value is always considered unsigned.

**Note:** The buffer should be large enough to contain **any** possible value: (sizeof(int)\*8+1) for radix=2, i.e. 33 bits.

### Parameters:

| Type | Name    | Description   |
|------|---------|---|
| int  | value   | The value to be represented as a string.                                      |
| char | *buffer | The buffer - where to store the resulting string.                             |
| int  | radix   | The numeral radix in which the value has to be represented, between 2 and 36. |

### Returns

A pointer to the string

## 20.24. ShowCounters

### Synopsis:

```
void ShowCounters( );
```

### Description:

This diagnostic function will dump all of the system counters to stdout.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 20.25. ForceReboot

### Required Header File:

```
#include <bsp.h>           // Found in C:\Nburn\include
```

### Synopsis:

```
void ForceReboot( );
```

### Description:

This function will reboot your NetBurner device.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 20.25.1. Example

### Mod5270, Mod5272, and/or Mod5282 only

```
#include "predef.h"
#include "..\mod5272\system\sim5272.h" // For Mod5272
                                   // For Mod5282 use: #include "..\mod5282\system\sim5282.h"
                                   // For Mod52270 use: #include "..\mod5282\system\sim5270.h"

#include <stdio.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpcclient.h>
#include <bsp.h>

extern "C"
{
    void UserMain(void * pd);
}

/*-----
   User Main
   -----*/

const char * AppName="Software Reset";

void UserMain(void * pd)
{
    InitializeStack();
    OSChangePrio(MAIN_PRI0);
    EnableAutoUpdate();
    iprintf("\r\nStarting NetBurner Software Reset Example v1.0\r\n");
    iprintf("-----\r\n");
    iprintf("To Software reset after 3 seconds press any key.\r\n\r\n");
    getchar();
    iprintf("\r\n");

    for(int i=0; i<3; i++)
    {
        OSTimeDly((WORD)(TICKS_PER_SECOND));
        iprintf("%d\r\n",i+1);
    }

    iprintf("\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n");

    ForceReboot(); // Reboot your Mod5272, Mod5270, or Mod5282 Module

    iprintf("This will NOT print!\r\n");
}
}
```

## 20.26. EtherLink

### Required Header File:

```
#include <ethernet.h>          // Found in C:\Nburn\include
```

### Synopsis:

```
BOOL EtherLink( );
```

### Description:

This function reports the status of the Ethernet link.

### Parameters:

None

### Returns:

True --- If the Ethernet link is valid  
False --- If there is no Ethernet connection

## 20.27. EtherSpeed100

### Required Header File:

```
#include <ethernet.h>          // Found in C:\Nburn\include
```

### Synopsis:

```
BOOL EtherSpeed100( );
```

### Description:

This function reports if the Ethernet link is operating at 100 MB.

### Parameters:

None

### Returns:

True --- If the Ethernet link is operating at 100 MB

**Important:** The EtherLink function **must** return **true** for this value to be **valid**.



## 20.28. EtherDuplex

### Required Header File:

```
#include <ethernet.h>      // Found in C:\Nburn\include
```

### Synopsis:

```
BOOL EtherDuplex( );
```

### Description:

This function reports if the Ethernet link is in Full Duplex mode.

### Parameters:

None

### Returns:

True --- If the Ethernet link is in full duplex mode

**Important:** The EtherLink function **must** return **true** for this value to be **valid**.

## 20.29. ManualEthernetConfig

### Required Header File:

```
#include <ethernet.h> // Found in C:\Nburn\include
```

### Synopsis:

```
void ManualEthernetConfig ( BOOL FullSpeed, BOOL FullDuplex, BOOL  
AutoNegotiate )
```

### Description:

This function controls the speed and duplex of the Ethernet connection. The default connection at boot will be auto-negotiated.

If this function is called with AutoNegotiate=**TRUE**, then the other two parameters (i.e. FullSpeed and FullDuplex) do **not** care, and the connection **will** re-establish itself as an auto-negotiation link.

If this function is called with AutoNegotiate=**FALSE** then set:

- FullSpeed=TRUE for 100BaseTX **or** FullSpeed=FALSE for 10BaseT
- 2FullDuplex=TRUE for Full Duplex **or** FullDuplex=FALSE for Half Duplex

**Note:** Setting both speed and duplex is usually **required** for establishing a connection to a device that does **not** support auto-negotiation.

### Parameters:

| Type | Name          | Description   |
|------|---------------|---|
| BOOL | FullSpeed     | Connection speed -- select if auto-negotiate disabled   |
| BOOL | FullDuplex    | Connection duplex -- select if auto-negotiate disabled. |
| BOOL | AutoNegotiate | Connect with auto-negotiate.                            |

### Returns:

Nothing --- This is a void function

## 21. TCP/IP Library

### 21.1. Introduction

The TCP/IP Stack is a high performance TCP/IP Stack for embedded applications. The TCP/IP Stack is integrated with the RTOS, Web Server, and I/O System providing easy development of network applications. The NetBurner Web Server is integrated with the TCP/IP Stack and RTOS, enabling you to quickly develop dynamic web pages and content.

This section covers the TCP/IP specific functions (i.e. the creation and setup of sockets) in the NetBurner system. **Important:** This section does **not** cover the **read/write operations** on these sockets. The read/write operations are covered in the **I/O System Library**. For **UDP** specific functions please refer to the **UDP Library**.

#### Required Header Files

```
#include <ip.h>           // Found in C:\Nburn\include
#include <tcp.h>           // Found in C:\Nburn\include
#include <udp.h>           // Found in C:\Nburn\include
#include <multihome.h>     // Found in C:\Nburn\include
```

#### Functions

##### IP Stack Start, Stop, and Add Functions

- InitializeStack --- Start the IP and TCP stack
- KillStack --- Shutdown the IP stack
- AddInterface --- (MULTIHOME) Add an additional IP interface to the system

##### Socket Creation Functions

- listen --- Start listening to accept connections
- accept --- Accept a connection on a listening socket
- connect --- Connect initiate a connection to another host
- connectvia --- Connect initiate a connection to another host via a specific interface  
(**Warning:** This only works with MULTIHOME)

##### Socket Option Functions

- setsockopt --- Set a socket option
- clrsockopt --- Clear a socket option
- getsockopt --- Get a socket option

## Get Socket Information Functions

- GetSocketRemoteAddr --- Get the IP address of the remote host associated with a socket
- GetSocketLocalAddr --- Get the IP address of local interface associated with a socket (Warning: This only works with MULTIHOMES)
- GetSocketRemotePort --- Get the remote port associated with a socket
- GetSocketLocalPort --- Get the local port associated with a socket

## Get Host by Name Function

- GetHostByName --- Look up an IP address using DNS

## Ping Functions

- Ping --- Sends an ICMP echo to an address and waits for a response
- PingViaInterface --- Sends an ICMP echo through a specified interface and waits for a response

## TFTP Functions

- GetTFTP --- Read a file from a TFTP server
- SendTFTP --- Send a file to a TFTP server

## Diagnostic Functions

- ShowArp --- Dump the ARP cache to stdio
- DumpTcpDebug --- Dump the TCP debugging log to stdio
- EnableTcpDebug --- Enable the TCP debug log
- ShowIPBuffer --- Dump a pool pointer to stdio, interpreting it as a packet
- GetFreeCount --- Get number of free buffer available
- ShowBuffer --- Show the raw data in a poolptr

## Byte Order Translation (not needed on Coldfire) Functions

- HTONS --- Translate a WORD from host order to network order
- HTONL --- Translate a DWORD from host order to network order
- NTOHS --- Translate a WORD from network order to host order
- NTOHL --- Translate a DWORD from network order to host order

## 21.2. InitializeStack

### Synopsis:

```
void InitializeStack( IPADDR ipaddr=0, IPADDR ipMask=0, IPADDR ipGate=0
);
```

### Description:

This function initializes the IP stack.

### Parameters:

| Type   | Name     | Description           |
|--------|----------|-----------------------|
| IPADDR | ipaddr=0 | The IP Address        |
| IPADDR | ipMask=0 | The IP (Network) Mask |
| IPADDR | ipGate=0 | The IP Gateway        |

**Important:** If **no** values are passed in for the addresses, the **default** values are **copied** from the **system configuration** record.

### Returns:

Nothing --- This is a void function

### See Also:

KillStack --- Shut down the IP stack

ConfigRecord --- The configuration storage structure

### Example Applications:

Simple Html --- Located by default in C:\Nburn\examples

tcp2Serial --- Located by default in C:\Nburn\examples

## 21.3. KillStack

### Synopsis:

```
void KillStack( );
```

### Description:

This function shuts down the IP stack.

### Parameters:

None

### Returns:

Nothing --- This is a void function

### See Also:

InitializeStack --- Start the IP and TCP stack

## 21.4. AddInterface (Multihome)

### Required Header File:

```
#include<multihome.h>          // Found in C:\Nburn\include
```

### Synopsis:

```
int AddInterface( IPADDR addr, IPADDR mask, IPADDR gateway );
```

### Description:

This function creates/initializes a new IP interface. This call adds a new interface to the system.

**Warning:** This call only works if you have defined the variable **MULTIHOME** in **C:\Nburn\include\predef.h** and rebuilt all of the system libraries.

### Parameters:

| Type   | Name    | Description                                |
|--------|---------|--|
| IPADDR | addr    | The IP Address of the new interface.       |
| IPADDR | mask    | The IP (Network) Mask of the new interface |
| IPADDR | gateway | The IP Gateway of the new interface.       |

### Returns:

Any value greater than 0 --- Equals the interface number  
-1 --- Failed too many interfaces

## 21.5. listen

### Synopsis:

```
int listen( IPADDR addr, WORD port, BYTE maxpend=5 );
```

### Description:

This function starts listening for connections on a TCP port. You must accept connections from this socket before you can use them. **Note:** You may use select to wait for connections on multiple listening sockets, by putting a listening socket in the readfds.

### Parameters:

| Type   | Name    | Description   |
|--------|---------|---|
| IPADDR | addr    | The address from which to accept connections.<br><b>Note:</b> If you want to accept connections from anywhere pass in the value INADDR_ANY. |
| WORD   | port    | The port to listen to.  |
| BYTE   | maxpend | The maximum number of pending connections to store on this listening socket.  |

### Returns:

A file descriptor for the listening socket

A negative number if there was an error:

- TCP\_ERR\_NOCON (-2) --- Indicates that you have attempted to read/write from a socket that does not have a connection established yet.
- TCP\_ERR\_NONE\_AVAIL (-5) --- Indicates that you have attempted to allocate a socket, but no socket is currently available.

### See Also:

accept --- Accept a connection on a listening socket

close --- Close open file descriptors (In the I/O section of this manual)

### Example Application:

tcp2serial --- Located by default in C:\Nburn\examples



## 21.6. accept

### Synopsis:

```
int accept( int listening_socket, IPADDR * address, WORD * port, WORD
timeout );
```

### Description:

This function accepts a connection from a listening socket.

### Parameters:

| Type   | Name             | Description   |
|--------|------------------|---|
| int    | listening_socket | The listening socket to accept from.  |
| IPADDR | *address         | A pointer to the IPADDR that will receive the IPADDR of the connection.<br><b>Note:</b> This parameter can be NULL.   |
| WORD   | *port            | A pointer to the WORD that will receive the port number of the connection.<br><b>Note:</b> This parameter can be NULL |
| WORD   | timeout          | The number of time ticks to wait for a connecting socket. <b>Note:</b> 0 waits forever.                               |

### Returns:

The file descriptor of the connected socket

A negative number if there was an error:

- TCP\_ERR\_TIMEOUT (-1) --- Indicates that the connection has timed out.
- TCP\_ERR\_NOCON (-2) --- Indicates that you have attempted to read/write from a socket that does not have a connection established yet.
- TCP\_ERR\_CLOSING (-3) --- Indicates that you have attempted to read/write from a socket that has already been closed.
- TCP\_ERR\_NOSUCH\_SOCKET (-4) --- Indicates that you have attempted to allocate a socket that does not exist.
- TCP\_ERR\_NONE\_AVAIL (-5) --- Indicates that you have attempted to allocate a socket, but no socket is currently available.
- TCP\_ERR\_CON\_RESET (-6) --- Indicates that you have attempted to read/write from a connection that has been reset by the other side.
- TCP\_ERR\_CON\_ABORT (-7) --- This is an internal error that the client won't usually see.

### See Also:

listen ---Start listening to accept connections.

close ---Close open file descriptors (In the I/O section of this manual)

## 21.7. connect

### Synopsis:

```
int connect( IPADDR addr, WORD localport, WORD remoteport, DWORD
timeout );
```

### Description:

This function connects to another host.

### Parameters:

| Type   | Name       | Description   |
|--------|------------|---|
| IPADDR | addr       | The address to connect to.  |
| WORD   | localport  | The local port to use for the connection.<br><b>Note:</b> A value of 0 causes the stack to select an unused port. |
| WORD   | remoteport | The remote port to connect to.  |
| DWORD  | timeout    | The number of time ticks to wait for a connecting socket. <b>Note:</b> A value of 0 waits forever.                |

### Returns:

The file descriptor of the connected socket

A negative number if there was an error:

- TCP\_ERR\_TIMEOUT (-1) --- Indicates that the connection has timed out.
- TCP\_ERR\_NOCON (-2) --- Indicates that you have attempted to read/write from socket that does not have a connection established yet.
- TCP\_ERR\_CLOSING (-3) --- Indicates that you have attempted to read/write from a socket that has already been closed.
- TCP\_ERR\_NONE\_AVAIL (-5) --- Indicates that you have attempted to allocate a socket, but no socket is currently available.
- TCP\_ERR\_CON\_RESET (-6) --- Indicates that you have attempted to read/write from a connection that has been reset by the other side.
- TCP\_ERR\_CON\_ABORT (-7) --- This is an internal error that the client won't usually see.

### See Also:

connectvia --- Initiate a connection to another host via a specific MULTIHOM interface

accept --- Accept a connection on a listening socket

listen --- Start listening to accept connections

close --- Close open file descriptors (In the I/O section of this manual)

## 21.8. connectvia

### Synopsis:

```
int connectvia( IPADDR addr, WORD localport, WORD remoteport, DWORD
timeout, IPADDR ipa );
```

### Description:

This function connects to another host via a specific IP interface.

### Parameters:

| Type   | Name       | Description  |
|--------|------------|--|
| IPADDR | addr       | The address to connect to.   |
| WORD   | localport  | The local port to use for the connection. <b>Note:</b> A value of 0 causes the stack to select an unused port. |
| WORD   | remoteport | The remote port to connect to.   |
| DWORD  | timeout    | The number of time ticks to wait for a connecting socket. <b>Note:</b> A value of 0 waits forever.             |
| IPADDR | ipa        | The IP address of the interface to use for making this connection.   |

### Returns:

The file descriptor of the connected socket

A negative number if there was an error:

- TCP\_ERR\_TIMEOUT (-1) --- Indicates that the connection has timed out.
- TCP\_ERR\_NOCON (-2) --- Indicates that you have attempted to read/write from socket that does not have a connection established yet.
- TCP\_ERR\_CLOSING (-3) --- Indicates that you have attempted to read/write from a socket that has already been closed.
- TCP\_ERR\_NONE\_AVAIL (-5) --- Indicates that you have attempted to allocate a socket, but no socket is currently available.
- TCP\_ERR\_CON\_RESET (-6) --- Indicates that you have attempted to read/write from a connection that has been reset by the other side.
- TCP\_ERR\_CON\_ABORT (-7) --- This is an internal error that the client will not (usually) see.

### See Also:

connect --- Initiate a connection to another host via default interface

accept --- Accept a connection on a listening socket

listen --- Start listening to accept connections

close --- Close open file descriptors (In the I/O section of this manual)

## 21.9. setsockopt

### Synopsis:

```
int setsockopt( int fd, int option );
```

### Description:

This function sets a socket option.

### Parameters:

| Type | Name   | Description                 |
|------|--------|-----------------------------|
| int  | fd     | The socket file descriptor. |
| int  | option | The option to set.          |

### Options:

SO\_NONAGLE --- Disables the NAGLE algorithm for this socket.

SO\_NOPUSH --- Does not send packets with every write. It holds the data for larger packets.

### Returns:

The bitmask of all the options for this socket

### See Also:

clrsocketoption --- Clears a specific socket option

getsockopt --- Get a socket option

## 21.10. clrsockoption

### Synopsis:

```
int clrsockoption( int fd, int option );
```

### Description:

This function clears a specific socket option.

### Parameters:

| Type | Name   | Description                 |
|------|--------|-----------------------------|
| int  | fd     | The socket file descriptor. |
| int  | option | The option to clear.        |

### Options:

SO\_NONAGLE --- Disables the NAGLE algorithm for this socket

SO\_NOPUSH --- Does not send packets with every write. It holds the data for larger packets

### Returns:

The bit of all enabled options

### See Also:

setsockoption --- Set a socket option

getsockoption --- Get a socket option

## 21.11.      getsockopt

### Synopsis:

```
int getsockopt( int fd );
```

### Description:

This function gets the current options for this socket.

### Parameter:

| Type | Name | Description                 |
|------|------|-----------------------------|
| int  | fd   | The socket file descriptor. |

### Options:

SO\_NONAGLE --- Disables the NAGLE algorithm for this socket.

SO\_NOPUSH --- Does not send packets with every write. It holds the data for larger packets.

### Returns:

The bit of all enabled options for the selected socket

### See Also:

setsockopt --- Set a socket option

clrsockopt --- Clear a socket option

## 21.12.      GetSocketRemoteAddr

### Synopsis:

```
IPADDR GetSocketRemoteAddr( int fd );
```

### Description:

This function gets the IP address of the remote host associated with this socket.

### Parameter:

| Type | Name | Description                 |
|------|------|-----------------------------|
| int  | fd   | The socket file descriptor. |

### Returns:

The IP Address of the remote host

### See Also:

GetSocketLocalAddr --- Get the IP address of local interface associated with a socket

GetSocketRemotePort --- Get the remote port associated with a socket

GetSocketLocalPort --- Get the local port associated with a socket

## 21.13. GetSocketLocalAddr

### Synopsis:

```
IPADDR GetSocketLocalAddr( int fd );
```

### Description:

This function gets the IP address of the interface associated with this socket.

**Warning:** This call only works if you have defined the variable **MULTIHOME** in **C:\Nburn\include\predef.h** and rebuilt all of the system libraries

### Parameter:

| Type | Name | Description                 |
|------|------|-----------------------------|
| int  | fd   | The socket file descriptor. |

### Returns:

The IP Address of the associated interface

### See Also:

GetSocketRemoteAddr --- Get the IP address of the remote host associated with a socket

GetSocketRemotePort --- Get the remote port associated with a socket

GetSocketLocalPort --- Get the local port associated with a socket



## 21.14. GetSocketRemotePort

### Synopsis:

**WORD** GetSocketRemotePort( int fd );

### Description:

This function gets the remote port associated with this socket.

### Parameter:

| Type | Name | Description                 |
|------|------|-----------------------------|
| int  | fd   | The socket file descriptor. |

### Returns:

This function returns the (WORD) remote port of the remote host.

### See Also:

GetSocketRemoteAddr --- Get the IP address of the remote host associated with a socket

GetSocketLocalAddr --- Get the IP address of local interface associated with a socket

GetSocketLocalPort --- Get the local port associated with a socket

## 21.15.      GetSocketLocalPort

### Synopsis:

```
WORD GetSocketLocalPort( int fd );
```

### Description:

This function gets the local port associated with this socket.

### Parameter:

| Type | Name | Description                 |
|------|------|-----------------------------|
| int  | fd   | The socket file descriptor. |

### Returns:

This function returns the (WORD) local port of the remote host.

### See Also:

GetSocketRemoteAddr --- Get the IP address of the remote host associated with a socket

GetSocketLocalAddr --- Get the IP address of local interface associated with a socket

GetSocketRemotePort --- Get the remote port associated with a socket

## 21.16. GetHostByName

### Required Header File:

```
#include <dns.h> // Found in C:\Nburn\include
```

### Synopsis:

```
int GetHostByName( const char * name, IPADDR * pIpaddr, IPADDR  
dns_server, DWORD timeout );
```

### Description:

This function looks up the IP address of the named host.

### Parameters:

| Type       | Name       | Description  |
|------------|------------|--|
| const char | *name      | The name to resolve (e.g. www.netburner.com).  |
| IPADDR     | *pIpaddr   | A pointer to the IPADDR variable. (Used to store the retrieved address.)                     |
| IPADDR     | dns_server | The IPADDR of the DNS server to query.<br><b>Note:</b> 0 uses the stored system default.     |
| WORD       | timeout    | The number of time ticks to wait for a response.<br><b>Note:</b> A value of 0 waits forever. |

### Returns:

DNS\_OK --- On Success

DNS\_TIMEOUT --- If the DNS Server does not respond

DNS\_NOSUCHNAME --- If the DNS Server can not find the name

DNS\_ERR --- If the received DNS response has errors

### See Also:

AsciiToIp --- Convert a dotted decimal IP string to an IP address

ShowIP --- Show an IP address as dotted decimal on stdio

## 21.17. Ping

### Synopsis:

```
int Ping( IPADDR to, WORD id, WORD seq, WORD maxwaitticks );
```

### Description:

This function "pings" the selected address and waits for a response. Ping (i.e. **P**acket **I**nternet **G**roper) is an Internet utility used to determine whether a particular IP address is online by sending out a packet and waiting for a response. Ping is also used to test and debug a network as well as see if a user is online. Ping can also function like a DNS (Domain Name Server) because "pinging" a domain name will return its IP address. The Ping function is used for the primary ethernet interface. **Note:** If you need to ping through another specified interface, you **must** use the **PingVialInterface** function.

### Returns:

The number of ticks the response took  
-1 --- If it timed out

### See Also:

PingVialInterface --- Sends an ICMP echo through a specified interface and waits for a response  
SendPing --- Sends an ICMP echo to an address

### Example:

```
/* This function pings the address given in buffer */
void ProcessPing(char * buffer)
{
    IPADDR addr_to_ping;
    char * cp=buffer; /* Trim leading white space */
    while ((*cp) && (isspace(*cp))) cp++; /* Get the address or use the
    default */
    if (cp[0])
        addr_to_ping=AsciiToIp(cp);
    else
        addr_to_ping=IpGate;
    printf("\nPingging :");
    ShowIP(addr_to_ping);
    printf("\n");
    int rv=Ping(addr_to_ping,1/*Id */,1 /*Seq */,100/*Max Ticks*/);
    if (rv== -1) printf(" Failed! \n");
    else
        printf(" Response Took %d ticks\n",rv);
}
```

## 21.18. PingViaInterface

### Synopsis:

```
int PingViaInterface( IPADDR to, WORD id, WORD seq, WORD maxwaitticks,  
int interface );
```

### Description:

This function "pings" the selected address through a specified interface and waits for a response. Ping (i.e. **P**acket **I**nternet **G**roper) is an Internet utility used to determine whether a particular IP address is online by sending out a packet and waiting for a response. Ping is also used to test and debug a network as well as see if a user is online. Ping can also function like a DNS (Domain Name Server) because "pinging" a domain name will return its IP address. **Note:** If pinging over WiFi (or if any other alternate IP interface is desired), then this function is **required** in lieu of the Ping function.

### Returns:

The number of ticks the response took  
-1 --- If it timed out

### See Also:

Ping --- Sends an ICMP echo to an address and waits for a response  
SendPing --- Sends an ICMP echo to an address

## 21.19. SendPing

### Synopsis:

```
void SendPing( IPADDR to, WORD id, WORD seq, int interface );
```

### Description:

This function "pings" the selected address. Ping (i.e. **P**acket **I**nternet **G**roper) is an Internet utility used to determine whether a particular IP address is online by sending out a packet and waiting for a response. Ping is also used to test and debug a network as well as see if a user is online. Ping can also function like a DNS (Domain Name Server) because "pinging" a domain name will return its IP address. The Ping function is used for the primary ethernet interface. **Note:** If you need to ping through another specified interface, you **must** use the **PingVialInterface** function.

### Returns:

Nothing --- This is a void function

### See Also:

Ping --- Sends an ICMP echo to an address and waits for a response

PingVialInterface --- Sends an ICMP echo through a specified interface and waits for a response

## 21.20. GetTFTP

### Synopsis:

```
int GetTFTP( PCSTR fname, PCSTR mode, PBYTE buffer, int & len, DWORD
timeout, IPADDR server, WORD opening_port );
```

### Description:

This function reads a file from a TFTP Server and put it in the passed in buffer. Important: TFTP (i.e. Trivial File Transfer Protocol) is a version of the TCP/IP FTP protocol that has no directory or password capability.

### Parameters:

| Type   | Name         | Description  |
|--------|--------------|--|
| PCSTR  | fname        | The name of the file to retrieve.  |
| PCSTR  | mode         | The opening mode: b (binary) or t (text).  |
| PBYTE  | buffer       | A pointer to the memory area to hold the received file.  |
| int    | &len         | A reference to the buffer length. It holds the maximum length when the function is called and holds the number of bytes actually copied on return. |
| DWORD  | timeout      | The number of ticks to wait for the operation to complete.   |
| IPADDR | server       | The IP Address to send the TFTP request to.  |
| WORD   | opening_port | The port to send the TFTP request to.  |

### Returns:

TFTP\_OK (0)  
TFTP\_TIMEOUT (1)  
TFTP\_ERROR (2)

### See Also:

SendTFTP --- Send a file to a TFTP server  
NBTFPT --- A TFTP Server for Win32

## 21.21. SendTFTP

### Synopsis:

```
int SendTFTP( PCSTR fname, PCSTR mode, PBYTE buffer, int & len, DWORD
timeout, DWORD pkttimeout, IPADDR server, WORD opening_port );
```

### Description:

This function sends a file to a TFTP server. Important: TFTP (i.e. Trivial File Transfer Protocol) is a version of the TCP/IP FTP protocol that has no directory or password capability.

### Parameters:

| Type   | Name         | Description   |
|--------|--------------|---|
| PCSTR  | fname        | The name of the file to put on the TFTP Server.   |
| PCSTR  | mode         | The opening mode: b (binary) or t (text).   |
| PBYTE  | buffer       | A pointer to the memory area that holds the file to be sent.  |
| int    | &len         | A reference to the buffer length. It holds the maximum length when the function is called, and holds the number of bytes actually copied on return. |
| DWORD  | timeout      | The total number of ticks to wait for the operation to complete.  |
| DWORD  | pkttimeout   | The number of ticks to wait for timeout on each packet sent.  |
| IPADDR | server       | The IP Address to send the TFTP request to.   |
| WORD   | opening_port | The port to send the TFTP request to.   |

### Returns:

TFTP\_OK (0)  
TFTP\_TIMEOUT (1)  
TFTP\_ERROR (2)

### See Also:

NBTFTP --- A TFTP Server for Win32  
GetTFTP --- Read a file from a TFTP server



## 21.22. ShowArp

### Synopsis:

```
void ShowArp( );
```

### Description:

This function dumps the ARP cache to stdio. ARP (**A**ddress **R**esolution **P**rotocol) is a protocol for mapping an Internet Protocol address (IP address) to a physical machine address that is recognized in the local network. The physical machine address is also known as a Media Access Control (MAC) address. A table, usually called the ARP cache, is used to maintain a correlation between each MAC address and its corresponding IP address. ARP provides the protocol rules for making this correlation and providing address conversion in both directions.

When an incoming packet destined for a host machine on a particular local area network arrives at a gateway, the gateway asks the ARP program to find a physical host or MAC address that matches the IP address. The ARP program looks in the ARP cache and, if it finds the address, provides it so that the packet can be converted to the right packet length and format and sent to the machine. If no entry is found for the IP address, ARP broadcasts a request packet in a special format to all the machines on the LAN to see if one machine knows that it has that IP address associated with it. A machine that recognizes the IP address as its own returns a reply so indicating. ARP updates the ARP cache for future reference and then sends the packet to the MAC address that replied.

Since protocol details differ for each type of local area network, there are separate ARP Requests for Comments (RFC) for Ethernet, ATM, Fiber Distributed-Data Interface, High-Performance Parallel Interface (HIPPI), and other protocols. There is also a Reverse ARP (RARP) for host machines that do not know their IP address. RARP enables them to request their IP address from the gateway's ARP cache.

### Parameters:

None

### Returns:

Nothing --- This is a void function

### See Also:

ShowIPBuffer --- Dump a pool pointer to stdio, interpreting it as a packet

## 21.23. DumpTcpDebug

### Synopsis:

```
void DumpTcpDebug( );
```

### Description:

This function dumps the TCP debugging log to stdio.

### Parameters:

None

### Returns:

Nothing ---This is a void function

### See Also:

EnableTcpDebug --- Enable the TCP debug log

## 21.24.      **EnableTcpDebug**

### **Synopsis:**

```
void EnableTcpDebug( WORD db );
```

### **Description:**

This function enables the TCP debug log.

### **Returns:**

Nothing --- This is a void function

### **See Also:**

DumpTcpDebug --- Dump the TCP debugging log to stdio

## 21.25. ShowIPBuffer

### Synopsis:

```
void ShowIPBuffer( PoolPtr rp );
```

### Description:

This function dumps a pool pointer to stdio, interpreting it as a packet.

### Returns:

Nothing --- This is a void function.

### See Also:

ShowBuffer --- Show the raw data in a pool pointer  
GetFreeCount --- Get number of free buffer available  
ShowArp --- Dump the ARP cache to stdio

## 21.26.      **GetFreeCount**

### **Synopsis:**

```
WORD GetFreeCount( );
```

### **Description:**

This function gets the number of free buffers available.

### **Parameters:**

None

### **See Also:**

ShowIPBuffer --- Dump a pool pointer to stdio, interpreting it as a packet  
ShowBuffer --- Show the raw data in a pool pointer

## 21.27. ShowBuffer

### Synopsis:

```
void ShowBuffer( PoolPtr p );
```

### Description:

This function shows the raw data in a pool pointer.

### Returns:

Nothing --- This is a void function

### See Also:

ShowIPBuffer --- Dump a pool pointer to stdio, interpreting it as a packet  
GetFreeCount --- Get number of free buffer available

## 21.28.     HTONS

### Synopsis:

```
WORD HTONS( WORD x );
```

### Description:

This function translates a WORD from host order to network order.

### See Also:

HTONL --- Translate a DWORD from host order to network order  
NTOHS --- Translate a WORD from network order to host order  
NTOHL --- Translate a DWORD from network order to host order

## 21.29. HTONL

### Synopsis:

```
DWORD HTONL( DWORD x );
```

### Description:

This function translates a DWORD from host order to network order.

### See Also:

HTONS --- Translate a WORD from host order to network order  
NTOHS --- Translate a WORD from network order to host order  
NTOHL --- Translate a DWORD from network order to host order



## 21.30. NTOHS

### Synopsis:

```
WORD NTOHS( WORD x );
```

### Description:

This function translates a WORD from network order to host order.

### See Also:

HTONL --- Translate a DWORD from host order to network order  
HTONS --- Translate a WORD from host order to network order  
NTOHL --- Translate a DWORD from network order to host order

## 21.31. NTOHL

### Synopsis:

```
DWORD NTOHL( DWORD x );
```

### Description:

This function translates a DWORD from network order to host order.

### See Also:

HTONL --- Translate a DWORD from host order to network order  
HTONS --- Translate a WORD from host order to network order  
NTOHS --- Translate a WORD from network order to host order

## 22. UDP Library

### Required Header File

```
#include <udp.h>           // Found in C:\Nburn\include
```

### Constructors and Destructor

- UDPPacket --- Construct a UDP object by waiting on a FIFO
- UDPPacket --- Make a UDP packet from a pool buffer
- UDPPacket --- Make an empty UDP packet
- ~UDPPacket --- UDP packet destructor

### Check Packet Validity

- Validate --- Returns true if the packet is valid

### Packet Element Access

- SetSourcePort --- Set the source port for the packet
- GetSourcePort --- Read source port
- SetDestinationPort --- Set the destination port
- GetDestinationPort --- Get the destination port

### Data Access Functions

- GetDataBuffer --- Get a pointer to the data buffer
- SetDataSize --- Set the size of the data section
- GetDataSize --- Get the size of the data section
- ResetData --- Zero the data buffer length

### Append Data Functions

- AddData --- Add data on the end
- AddData --- Add a zero terminated string
- AddDataWord --- Add a WORD
- AddDataByte --- Add a Byte

### Pool Pointer Access Functions

- ReleaseBuffer --- Release the UDP objects captive buffer
- GetPoolPtr --- Get a handle to the UDP objects captive buffer

## Send Functions

- SendAndKeep --- Send a copy of the attached pool pointer
- SendAndKeepVia --- Send a copy of the attached pool ptr via a specific interface
- Send --- Send and free the attached pool buffer
- SendVia --- Send and free the attached pool buffer via a specific interface

## Related Functions

- RegisterUDPFifo --- Register to listen to a specific UDP port
- UnregisterUDPFifo --- Unregister a listening UDP Fifo

## 22.1. UDPPacket (FIFO)

### Synopsis:

```
UDPPacket( OS_FIFO * fifo, DWORD wait );
```

### Description:

This function creates a UDP packet from a FIFO. The routine will wait pending a FIFO return.

**Note:** If the FIFO times out, then an invalid UDP packet will be created. This function verifies that the UDP packet is valid with the Validate function.

### Returns:

The UDP Packet - the verified returned packet with the Validate function

## 22.2. UDPPacket (Pool Buffer)

### Synopsis:

```
UDPPacket( PoolPtr p );
```

### Description:

This creates a UDP packet from a pool buffer.

### Returns:

The UDP Packet - the verified returned packet with the Validate function

## 22.3. UDPPacket

### Synopsis:

```
UDPPacket( );
```

### Description:

This function creates a UDP packet with no data.

### Parameters:

None

### Returns:

No value returned

## 22.4. ~UDPPacket

### Synopsis:

```
~UDPPacket( );
```

### Description:

This function is the UDPPacket destructor.

### Parameters:

None

### Returns:

No value returned



## 22.5. Validate

### Synopsis:

```
BOOL Validate( );
```

### Description:

This is a Boolean validation function. **Important:** This function should **only** be called when receiving a UDP packet, because packets **are** set up correctly when they are **created**.

### Parameters:

None

### Returns:

True --- If the packet is valid

False --- If you try to validate a packet before it has been sent

## 22.6. SetSourcePort

### Synopsis:

```
void SetSourcePort( WORD );
```

### Description:

This function sets the UDP Packet Source Port.

### Returns:

Nothing --- This is a void function

## 22.7. GetSourcePort

### Synopsis:

```
WORD GetSourcePort( );
```

### Description:

This function returns a UDP packet's source port number.

### Parameters:

None

### Returns:

The 16-bit source port number

## 22.8. SetDestinationPort

### Synopsis:

```
void SetDestinationPort( WORD );
```

### Description:

This function sets the UDP Packet destination port number.

### Returns:

Nothing --- This is a void function

## 22.9. GetDestinationPort

### Synopsis:

```
WORD GetDestinationPort( );
```

### Description:

This function returns a UDP packet's destination port number.

### Parameters:

None

### Returns:

The 16-Bit destination port number

## 22.10.      **GetDataBuffer**

### **Synopsis:**

```
PBYTE GetDataBuffer( );
```

### **Description:**

This function gets a pointer to the data buffer.

### **Parameters:**

None

## 22.11.      **SetDataSize**

### **Synopsis:**

```
void SetDataSize( WORD );
```

### **Description:**

This function sets the size of the data section.

### **Returns:**

Nothing --- This is a void function

## 22.12.      **GetDataSize**

### **Synopsis:**

```
WORD GetDataSize( );
```

### **Description:**

This function gets the size of the data section.

### **Parameters:**

None



## 22.13.     ResetData

### Synopsis:

```
void ResetData( );
```

### Description:

This function zero's the data buffer length.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 22.14.     AddData

### Synopsis:

```
void AddData( PBYTE pData, WORD len );
```

### Description:

This function adds data on the end.

### Returns:

Nothing --- This is a void function

## 22.15.      AddData (Add a Zero Terminated String)

### Synopsis:

```
void AddData( PCSTR pData );
```

### Description:

This function adds a zero terminated string.

### Returns:

Nothing --- This is a void function

## 22.16.     AddDataWord

### Synopsis:

```
void AddDataWord( WORD w );
```

### Description:

This function adds a word.

### Returns:

Nothing --- This is a void function

## 22.17.     AddDataByte

### Synopsis:

```
void AddDataByte( BYTE b );
```

### Description:

This function adds a byte.

### Returns:

Nothing --- This is a void function

## 22.18. ReleaseBuffer

### Synopsis:

```
void ReleaseBuffer( );
```

### Description:

This function releases the UDP objects captive buffer.

### Parameters:

None

### Returns:

Nothing --- This is a void function

## 22.19.     **GetPoolPtr**

### **Synopsis:**

```
PoolPtr GetPoolPtr( );
```

### **Description:**

This function gets a handle to the UDP objects captive buffer.

### **Parameters:**

None

## **22.20.      SendAndKeep**

### **Synopsis:**

```
void SendAndKeep( IPADDR to, BYTE ttl );
```

### **Description:**

This function sends a copy of the attached pool pointer.

### **Returns:**

Nothing --- This is a void function



## 22.21.      **SendAndKeepVia**

### **Synopsis:**

```
void SendAndKeepVia( IPADDR to, IPADDR from_ip, BYTE ttl );
```

### **Description:**

This function sends a copy of the attached pool pointer via a specified interface.

### **Returns:**

Nothing --- This is a void function

## 22.22. Send

### Synopsis:

```
void Send( IPADDR to, BYTE ttl=0 );
```

### Description:

This function sends and frees the attached pool buffer.

### Returns:

Nothing --- This is a void function

## 22.23. SendVia

### Synopsis:

```
void SendVia( IPADDR to, IPADDR from_ip, BYTE ttl );
```

### Description:

This function sends and frees the attached pool buffer via the specified interface.

### Returns:

Nothing --- This is a void function

## 22.24. RegisterUDPFifo

### Synopsis:

```
void RegisterUDPFifo( WORD dest_port, OS_FIFO *pfifo );
```

### Description:

This function will register to listen to a specific UDP port.

### Returns:

Nothing --- This is a void function

## 22.25. UnregisterUDPFifo

### Synopsis:

```
void UnregisterUDPFifo( WORD destination_port );
```

### Description:

This function will unregister a listening UDP FIFO.

### Returns:

Nothing --- This is a void function